AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation

ABSTRACT

This report presents a Common Foundation for Aspect-Oriented Software Development. A Common Foundation is required to enable effective communication and to enable integration of activities within the Network of Excellence. This Common Foundation is realized by developing an ontology, i.e. the shared meaning of terms and concepts in the domain of AOSD. In the first part of this report, we describe the definitions of an initial set of common AOSD terms. There is general agreement on these definitions. In the second part, we describe the Common Foundation task in detail.

Document ID: AOSD-Europe-UT-01
Deliverable No: D9
Type: Report
Status: FINAL
Version: 1.0
Date: 27 May 2005
Author(s): Klaas van den Berg (University of Twente),
Jose Maria Conejero (University of Extremadura,
visiting researcher at the University of Twente),
Ruzanna Chitchyan (University of Lancaster)

Document History

Date	Version	Description	
21-04-2005	0.01	First set up of this document	
26-04-2005	0.15	Submitted for first review	
18-05-2005	0.50	Version circulated for reviews and comments.	
18-05-2005	0.52	Submitted for approval	
27-05-2005	1.00	Final version	

Contents

Executive Summary **9**
PART A: The Common AOSD Terminology **10**
1. AOSD Glossary **10**
1.1. Introduction 10
1.2. Definitions of common AOSD terms 12
1.3. Conclusion 14
PART B: The Common Foundation Task **15**
2. Introduction **15**
2.1. Goals 15
2.2. General considerations 16
2.3. Overview of the report 16
2.4. Milestones in the Common Foundation task 17
2.5. How to read this report 17
3. Development Process of a Common Foundation **19**
3.1. Introduction 19
3.2. Development Phases 19
3.2.1 Requirements 19
3.2.2 Design 20
3.2.3 Implementation 22
3.3. Collaborative Approach 23
3.3.1 Procedure 23
3.3.2 Activities in the development of the Common Foundation 25
3.4. Infrastructure 25
3.4.1 Tool Support 26
3.4.2 Model, View and Conceptual Framework 26
3.4.3 Modelling Taxonomies 28
3.5. Conclusion 28
4. AspectJ Terms and Concepts **29**
4.1. Introduction 29
4.2. AspectJ Glossary 29
4.3. AspectJ Taxonomy 30
4.3.1 Views of AspectJ Taxonomy 30
4.3.2 Software System View 31
4.3.3 Concern View 31
4.3.4 Crosscutting Design View 32
4.3.5 Aspect View 33
4.3.6 Pointcut Signature View 34
4.3.7 Other Views 35
4.3.8 Analysis of AspectJ Views 35
4.4. Conclusion 36

5. ComposeStar Terms and Concepts **37**
5.1. Introduction 37
5.2. ComposeStar Glossary 37
5.3. ComposeStar Taxonomy 38
5.3.1 Views of ComposeStar Taxonomy 38
5.3.2 Concern View 39
5.3.3 Software System View 40
5.3.4 Superimposition View 40
5.4. Conclusion 41
6. Development of Common AOSD Terminology **42**
6.1. Introduction 42
6.2. Selection of core terms 43
6.3. Glossary 44
6.4. Conclusion 47
7. Proposal for a Conceptual Framework for Crosscutting **48**
7.1. Introduction 48
7.2. Crosscutting 48
7.3. Conclusion 50
8. Conclusion **51**
Acknowledgement **54**
References **55**
Appendix A. Glossary with Definition of Ontology Terms **58**
Appendix B. Glossary with Common AOSD Terminology **60**
Appendix C. AspectJ - Glossary **61**
Advanced Terms and Definitions 63
Basic Terms and Definitions 66
General Terms and Definitions 68
References 70
Appendix D. ComposeStar - Glossary **71**
Advanced Terms and Definitions 71
Basic Terms and Definitions 71
General Terms and Definitions 74
References 74
Appendix E. AspectJ - Taxonomy Views **75**
AspectJ - Advice View 76
AspectJ - Aspect Association View 77
AspectJ - Aspect View 78
AspectJ - Concern View 79
AspectJ - Conditional Pointcut View 80
AspectJ - Crosscutting Design View 81
AspectJ - Crosscutting Implementation View 82
AspectJ - Dynamic Pointcut View 83
AspectJ - Join Point Model View 84
AspectJ - Kinded Pointcut View 85
AspectJ - Pointcut Signature View 86

AspectJ - Pointcut View 87
AspectJ - Software System View 88
AspectJ - Static Pointcut View 89
AspectJ - Weaving View 90

Figures

Figure 1. Iterative development of the Common Foundation for AOSD............................ 11
Figure 2. Relation between Taxonomy and Glossary (fragment of CWM [13]) 21
Figure 3. Simplified Workflow of Modification Requests 23
Figure 4. The development process in the collaborative approach [24] 24
Figure 5. Activities in the development of the Common Foundation................................ 25
Figure 6. Tool Support for Collaborative Ontology Development 26
Figure 7. Ontology Concepts with Views (modified version of Figure 2) 27
Figure 8. Model conventions for associated and specialized concepts 28
Figure 9. Model of Views of the AspectJ Taxonomy 30
Figure 10. Software System View of the AspectJ Taxonomy 31
Figure 11. Concern View of the AspectJ Taxonomy 32
Figure 12. Crosscutting Design View of the AspectJ Taxonomy 33
Figure 13. Aspect View of the AspectJ Taxonomy 34
Figure 14. Pointcut Signature View of the AspectJ Taxonomy 34
Figure 15. ComposeStar Terms per Category 38
Figure 16. View Model of ComposeStar Domain 39
Figure 17. Concern View of the ComposeStar Taxonomy 39
Figure 18. Software System View of the ComposeStar Taxonomy 40
Figure 19. Superimposition View of the ComposeStar Taxonomy 40
Figure 20. Common Terminology derived from different perspectives 43
Figure 21. Crosscutting of modules 49
Figure 22. AspectJ concepts and views (UML Package Diagram) 62

Tables

Table 1. Initial and Preferred Definitions of Common AOSD Terms. 14
Table 2 Summary statistics of AspectJ Glossary 30
Table 3. Number of Concepts per View in AspectJ Domain 35
Table 4 Summary statistics of ComposeStar Glossary 38

Abbreviations

AO	Aspect Orientation
AOSD	Aspect Oriented Software Development
CF	Common Foundation
CWM	Common Warehouse Metamodel
IA	Integration Activity
FR	Functional Requirement
MDA	Model Driven Architecture
MR	Modification Request
NFR	Non-Functional Requirement
NoE	Network of Excellence
OMG	Object Management Group
SOC	Separation of Concerns
UML	Unified Modelling Language
WP	Work Package

Say what you mean
Mean what you say
Don't say it mean [1]

Executive Summary

The goal of the Common Foundation task in the AOSD-Europe project is to provide an ontology of aspect-oriented concepts. The common foundation is a critical success factor for the project: *Sharing a common terminology and associated conceptual model is a key contributing factor to effective technical discussions*.

The task was carried out from September 2004 until May 2005 under supervision of the University of Twente, with major contributions from the University of Lancaster, and further from other network partners, among others the University of Leuven.

The report has two parts:
- Part A describes the definitions of an initial set of common AOSD terms. There is general agreement on these definitions.
- Part B described the Common Foundation task in detail.

We describe a disciplined process for the development of the common foundation based on ontology engineering. We follow a collaborative approach and describe two parts of the ontology: a glossary with definitions of terms (terminology), and a taxonomy with concepts and their relations (conceptual domain model). We present requirements for an AOSD ontology.

We describe the products (glossaries, taxonomies) from this task. The products are the following: a taxonomy and glossary for AspectJ, a taxonomy and glossary for ComposeStar, a glossary for common AOSD terms, and a proposal for a conceptual framework for crosscutting (which could be part of the taxonomy for AOSD). A process related product from this task is the web-based infrastructure to support the ontology development.

The ontology described in this report is the first public version. It should evolve during the project based on research in the various labs. In order to facilitate this process, it is recommended that each deliverable in the project provides a glossary with definitions of terms. Based on these glossaries, there can be a yearly update of the ontology for the Common Foundation.

[1] Lyric 'Say what you mean' by Lunachick in their album Luxury Problem

PART A: The Common AOSD Terminology

1. AOSD Glossary

In this chapter, we describe the definitions of an initial set of core terms in AOSD. The initial definitions are based on the AOSD book by Filman et al. (2005). After several reviews, we selected preferred definitions that are more general, and for which there is general agreement.

1.1. Introduction

The goal of the Common Foundation task in the AOSD-Europe project is to provide an ontology of aspect-oriented concepts. A Common Foundation is required to enable effective communication and to enable integration of activities within the Network of Excellence.

"*The common foundation is aiming to achieve wide consensus that the terms and their definitions are acceptable (even though individuals might prefer other terms and definitions)*" [3].

Core terms

In a Workshop on AOSD Terminology at the University of Twente, we selected an initial set of core terms for the common glossary (the process is described in Chapter 6). These terms are the following:

- Separation of Concerns, Tyranny of Dominant Decomposition, Composition, Weaving, Decomposition, Modularization
- Concern, Crosscutting Concern, Crosscutting, Scattering, Tangling
- Aspect, Advice, Pointcut, Join Point, Join Point Model

This initial set with terms can be augmented in later phases of the AOSD-Europe project. We based our initial definitions on the Introduction (Chapter 1) in the book by Filman et al. (2005) [18]. We collected comments on these definitions in several review rounds by email. The result is a list of terms (Table 1) with definitions with the highest preference (see section 1.2).

Iterative development

We would like to stress that developing consensus on terminology is an iterative process. This report describes a first version of definitions. There should be regular updates, based on comments and new developments.

Moreover, the terminology should be made consistent with conceptual models of the AO domain. A first attempt to set up (part of) such a conceptual model is described in Part B of this report. These conceptual models should also evolve iteratively during the AOSD-Europe project.

Current state
In the Figure 1 we show schematically the current situation of the Common Foundation, with common terminology (T), conceptual models (M) and deployment (D) of terminology and models.

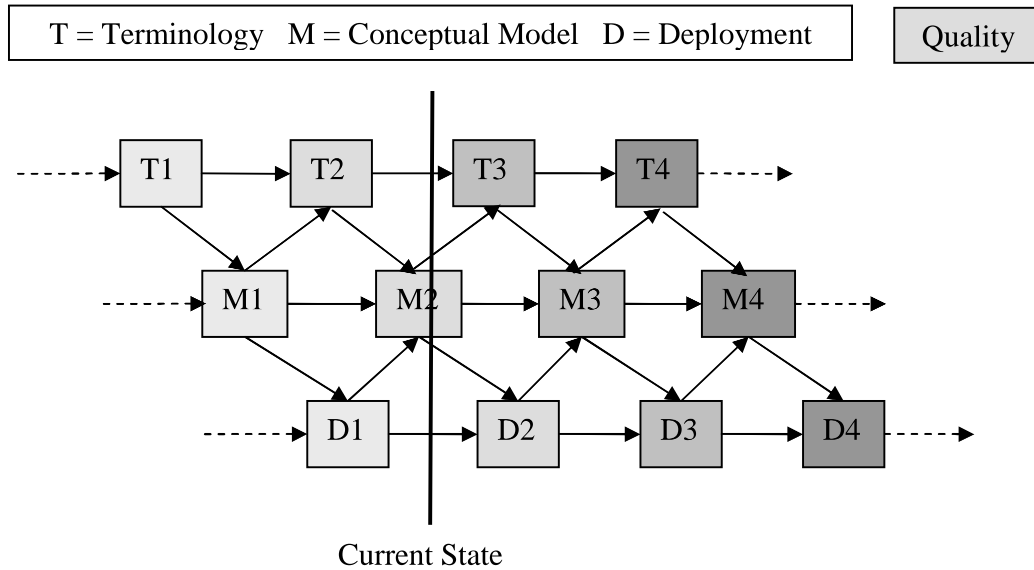


Figure 1. Iterative development of the Common Foundation for AOSD

The deployment of terminology and conceptual models should take place in the other work packages and labs, by giving concrete examples of the use of terms and concepts. The relation between terminology (glossary) and conceptual model (taxonomy) is given in section 3.2.2.

The *quality* of terminology, models and deployment is determined by among others: general acceptance, comprehensiveness, consistency, and preciseness. Moreover, terms and concepts should be applicable over different phases in the development lifecycle and for different aspect paradigms. A detailed description of quality requirements is given in section 3.2.1. The darkening of the shading in Figure 1 indicates increasing quality.

In the current situation - for the deliverable D9 - we iterated over some versions of terminology and reached reasonable agreement on the definition of core terms, we have partial conceptual models (described in Part B of this report), and only few applications of the use of terminology and conceptual models.

1.2. Definitions of common AOSD terms

In this section, we give the core terms in the AOSD-glossary with their definitions: the first definition is usually based on the Introduction (Chapter 1) in the book by Filman et al. (2005) [18], and the second definition (shaded grey) has the highest preference from the alternatives. In Chapter 6 we give an overview of alternative definitions (with lower preference).

Nr	**Term**	**Definition** (preferred definition in grey fields)	
1.	Separation of Concerns	Separation of concerns simplifies system development by allowing the development of specialized expertise and by producing an overall more comprehensible arrangement of elements. [18]	
2.		Separation of Concerns is an in depth study and realisation of concerns in isolation for the sake of their own consistency (adapted from “On the Role of Scientific Thought” by Dijkstra, EWD 447).	
3.	Tyranny of Dominant Decomposition	No explicit definition in Filman et al. [18]	
4.		The Tyranny of the Dominant Decomposition refers to restrictions (or tyranny) imposed by the selected decomposition technique (i.e. the dominant decomposition) on software engineer's ability to modularly represent particular concerns.	
5.	Composition	Composition is bringing together separately created software elements. [18]	
6.		Composition is the integration of multiple modular artefacts into a coherent whole.	
7.	Weaving	Weaving is the process of composing core functionality modules with aspects, thereby yielding a working system. [18]	
8.		Weaving: Historically this term is used to refer to the composition of aspects with other concerns in the system. See composition.	
9.	Decomposition	No explicit definition in Filman et al. [18]	
10.		Decomposition is the breaking down of a larger problem into a set of smaller problems which may be tackled individually.	

11.	Modularization	No explicit definition in Filman et al. [18]	
12.		No explicit definition required: available is standard SE literature	
13.	Module	No explicit definition in Filman et al. [18]	
14.		Synonym: Modular Artefact	
15.		No explicit definition required: available is standard SE literature	
16.	Concern	A concern is a thing in an engineering process about which it cares. [18]	
17.		A concern is an interest, which pertains to the system's development, its operation or any other matters that are critical or otherwise important to one or more stakeholders.	
18.	Crosscutting Concern	A crosscutting concern is a concern for which the implementation is scattered throughout the rest of an implementation. [18]	
19.		A crosscutting concern is a concern, which cannot be modularly represented within the selected decomposition. Consequently, the elements of crosscutting concerns are scattered and tangled within elements of other concerns.	
20.	Crosscutting	Crosscutting is a property of a concern for which the implementation is scattered throughout the rest of an implementation. [18]	
21.		Crosscutting is the scattering and/or tangling of concerns arising from the inability of the selected decomposition to modularise them effectively.	
22.	Scattering	No explicit definition in Filman et al. [18]	
23.		Scattering is the occurrence of elements that belong to one concern in modules encapsulating other concerns.	
24.	Tangling	Tangling occurs when the code for the implementation of concerns is intermixed. [18]	
25.		Tangling is the occurrence of multiple concerns mixed together in one module.	
26.	Aspect	An aspect is a modular unit designed to implement a concern. [18]	
27.		An aspect is a unit for modularising an otherwise crosscutting concern.	

28.	Advice	An advice is the behaviour to execute at a join point. [18]	
29.		An advice is an aspect element, which augments or constrains other concerns at join points matched by a pointcut expression.	
30.	Pointcut (Designator)	A Pointcut Designator describes a set of join points. [18]	
31.		A pointcut is a predicate that matches join points. More precisely, a pointcut is a relationship from JoinPoint -> boolean, where the domain of the relationship is all possible join points.	
32.	Join Point	A Join Point is a well-defined place in the structure or execution flow of a program where additional behaviour can be attached. [18]	
33.		A join point is a point of interest in some artefact in the software lifecycle through which two or more concerns may be composed.	
34.	Join Point Model	A Join Point Model (the kind of join points allowed) provides the common frame of reference to enable the definition of the structure of aspects. [18]	
35.		A join point model defines the kinds of join points available and how they are accessed and used.	

Table 1. Initial and Preferred Definitions of Common AOSD Terms.

1.3. Conclusion

In the previous section, we listed core terms in AOSD, and their definitions with the highest preference among partners of the AOSD-Europe project. The current version of the preferred common AOSD glossary is summarized alphabetically in Appendix B.
Discussion points are among others, the definition and granularity of modules (modular artefacts) in decompositions, as used in the preferred definitions, across different phases of the development lifecycle. Moreover, there should be a precise description of the relation between crosscutting, tangling and/or scattering.
After publishing this first version, there should be regular (e.g., yearly) updates, based on comments and new developments. This should be part of the iterative development as described in section 1.1.

PART B: The Common Foundation Task

2. Introduction

In this chapter, we summarize the goals and approach in the development of the AOSD Common Foundation. Then we give an overview of the topics described in this second part of the document.

2.1. Goals

The development of a Common Foundation for Aspect-Oriented Software Development (CF-AOSD) is one of the integration activities (IA1.2) in the Harmonization work package (WP1) of AOSD-Europe [3]. This activity is described as follows:

Objectives

AOSD is still a young field, most of the terms and concepts in use have been loosely defined, and often there is no commonly accepted term for a general concept. In fact, most of the terminology and the technology have been adopted from the AspectJ language. While acknowledging both the technical contributions and the pioneering role of AspectJ to make AO technology accessible to practitioners, the time has come to establish a broader foundation than that afforded by an aspect-oriented programming language alone.

In addition, we have identified two critical success factors for the proposed network:

- Establishing solid communication among its members: sharing a common terminology and associated conceptual model is a key contributing factor to effective technical discussions.
- A common foundation is important to ensure that results from the various activities, which take place in parallel, can be integrated.

To address these concerns, the common foundation task will define an ontology of aspect-oriented concepts. An ontology defines the (shared) meaning of a set of concepts for a given domain. This can be used to improve communication and interaction among people, or even among systems. Typically, ontologies consist of (often textual) definitions of concepts with precise (formal) specifications of their interdependencies.

The ontology is expected to have an important core about aspect-oriented models, but should also address areas such as requirements, software architectures, etc. From the point of view of the exploitation of results of the NoE, it is expected that a solid ontology, proposed and supported by major players in the field, is likely to become a defacto standard, and referred to in many contexts. As such, it will contribute to the visibility and impact of the NoE.

Deliverables

To address these concerns, the common foundation task will, as early as possible, define a draft ontology, prepared by a small, yet broad, task force. This draft must be discussed and adapted, aiming to achieve wide consensus that the terms and their definitions are acceptable (even though individuals might prefer other terms and definitions). After publishing the first version, there should be regular (e.g., yearly) updates, based on comments and new developments. Hence the deliverables of this activity are:

- Proposal for an "Ontology of Aspect-Orientation";
- "Ontology of Aspect-Orientation" –version 1.0;
- Regular new versions of the "Ontology of Aspect-Orientation".

2.2. General considerations

From the work package description in the previous section, we highlight the following issues:

- The common terminology should not be bound to specific aspect languages or life cycle phases.
- The common foundation should be realized by means of an ontology.
- There should be reasonable consensus about this ontology, achieved by discussion and reviews.
- There should be regular updates in order to improve the ontology according to new developments and new insights.
- The ontology should get the status of a de facto standard.

In the following section, we give an outline of the report.

2.3. Overview of the report

The chapters in this report are following merely the *chronological* order of the activities in the development process, although not all activities were carried out fully sequentially.

In chapter 3, we describe the approach to ontology engineering, the collaborative approach to ontology development, the requirements for an AOSD ontology, the distinction between glossary (terminology) and taxonomy (conceptual domain model), and the web-based infrastructure.

In chapter 4, we describe glossary of AspectJ terms (terminology) and a taxonomy of AspectJ concepts (conceptual model). The initial version of the glossary is based on a pilot study. Moreover, we applied the notation for conceptual modelling.

In chapter 5, we describe a first version of a glossary of ComposeStar terms (terminology) and a taxonomy of ComposeStar concepts (conceptual model).

In chapter 6, we describe a glossary of common AOSD terms (terminology). The initial version is based on terminology used in the book Filman et al. (2005). The terminology has been generalized, and alternative definitions are presented.

In chapter 7, we describe a proposal for a conceptual model of crosscutting. In the framework we propose a clear distinction between crosscutting, scattering and tangling.

In chapter 8, we give a summary of the report, we list some conclusions based on this Common Foundation task, and some recommendations.

The chronology of topics in these chapters has gone hand in hand with some events, which will be described in the following section.

2.4. Milestones in the Common Foundation task

The following (internal) milestones were present in the Common Foundation task:

- Internal Report (UTwente) on the Approach to a Common Foundation for Aspect Oriented Software Development (20 September 2004)
- Kick-off Meeting AOSD-Europe in Lancaster (21-22 September 2004)
  Presentation of the collaborative ontology approach and the ontology design with glossary and taxonomy.
- Pilot Study for Infrastructure and AspectJ terminology (14 October - 30 November 2005)
- Milestone M1.2 (30 November 2004)
  Report on the web-based infrastructure and the results of pilot study on AspectJ.
- Milestone M1.3 (26 February 2005)
  Report of AspectJ Glossary and Taxonomy and ComposeStar Glossary and Taxonomy.
- Workshop AOSD-Europe in Darmstadt (7-8 March 2005)
  Presentation of the state of the Common Foundation task.
- Terminology AOSD Workshop in Enschede (13 April 2005)
  Workshop with video conferencing about common terminology and the conceptual framework.
- Review Draft of D6 in WP1 Harmonization (started 26 April 2005)
  Comments on definition of AspectJ based terms and on using new terminology in the conceptual framework for crosscutting.
- Review of Draft Common Terminology in WP1 Harmonization (started 4 May 2005)
  A number of reviews are compiled in the final deliverable.
- Deliverable D9 to Approval Committee (18 May 2005)
- Final version of Deliverable D9 to AOSD-Europe project (27 May 2005)

The results of these events are documented in this report.

2.5. How to read this report

This is a long report and not all readers need to read the whole report. Each chapter starts with a description of the *focus* of the chapter.
You may read this report from different perspectives with emphasis on corresponding chapters. The *perspectives* could be the following:

1. **Perspective of the Common AOSD Terminology.**
   From this perspective, you should read about commonality analysis and the review process resulting in list with definition of terms in Chapter 6. A summary in presented in Chapter 1.

2. **Perspective of Aspect Languages**
   From this perspective, you should read about the AspectJ Glossary and Taxonomy in Chapter 4, and the ComposeStar Glossary and Taxonomy in Chapter 5

3. **Perspective of Foundations for AOSD**
   From this perspective, you should read about the proposal for a Conceptual Framework for Crosscutting in Chapter 7.

4. **Perspective of Ontology Development**
   From this perspective, you should read about the Taxonomy/Glossary Framework and Collaborative Approach in Chapter 3.

5. **Perspective of Deliverables for the project AOSD-Europe Network of Excellence**
   From this perspective, you should read about Development Process with Activities in Chapter 3 and the Conclusions and Recommendations in Chapter 8.

6. **Perspective of Common Foundation task**
   From this perspective, you should read the whole report carefully: the process part and the products part. The most important issue here is the further improvement of the common glossary and taxonomy.

In the following chapter, we start with a description of the approach and development process for a Common Foundation of AOSD.

3. Development Process of a Common Foundation

In this chapter, we summarize the approach for the development of the AOSD Common Foundation. It is based on a methodology for ontology engineering. We describe the distinction between glossary and taxonomy. Furthermore, we introduce the collaborative approach used in this process.

3.1. Introduction

We need a disciplined process for the development of the common foundation. We should set requirements for the AOSD ontology. We have to determine the notation to be used in the ontology and how the development of the ontology is supported with tools. Furthermore, we need to structure different subactivities in this Common Foundation task.

Overview of chapter

We start this chapter with de description of the requirements for an ontology, the design and the implementation (section 3.2). Then we describe the collaborative approach to ontology development and the related activities (section 3.3). We conclude with a description of the web-based support and the notation conceptual for domain models (section 3.4).

3.2. Development Phases

In the development of the Common Foundation (CF) we distinguish the following phases in the ontology development process: requirements for the ontology, design of the ontology, implementation of the ontology, validation and deployment of the ontology. There are supposed to be several iterations over these phases. We discuss the requirements, design and implementation of the ontology.

3.2.1 Requirements

In this section we list a number of requirements for the Common Foundation (CF). The most important requirements are:

Req 1. The CF shall comprehensively represent common terminology and concepts used in AOSD (descriptive standard).

Req 2. The CF shall be generally acceptable.

Other requirements are:

Req 3. The CF shall be accurate, complete, conflict-free, non-redundant

Req 4. The CF shall be unambiguous, verifiable, traceable

Req 5. The CF shall be usable: understandable, learnable, concise, accessible

Req 6. The CF shall be maintainable: analyzable, changeable (versions), testable, stable,

The first requirement Req 1 states the main characteristic of the Common Foundation as a descriptive standard. The CF is not meant to be a prescriptive and normative standard. These three types of standards can be described as follows [43]:

- *Descriptive*: give definitions of facts

- *Normative*: provide guidelines to be used as a basis for measurement, comparison or decision
- *Prescriptive*: define a particular way of doing something

Requirement Req 2 states the main non-functional requirement: the Common Foundation must be acceptable for stakeholders in AOSD-Europe [3] in order to facilitate communication between the partners and (re)use of terminology.
- *Generally accepted* means that the knowledge and practices described are applicable to most projects most of the time, and that there is widespread consensus about their value and usefulness. Generally accepted does not mean that the knowledge and practices described are or should be applied uniformly on all projects" (adapted from Project Management Body of Knowledge [39]).

The characteristics of Req 3 are described by Shanks et al. [42] for validating conceptual models.
- *Accuracy*. The model should accurately represent the semantics of the domain as perceived by the focal stakeholder(s);
- *Completeness*. The model should completely represent the semantics of the domain as perceived by the focal stakeholder(s);
- *Conflict-free*. The semantics represented in different parts of the model should not contradict one another (also called consistency)
- *No redundancy*. To reduce the likelihood of conflicts arising if and when the model is subsequently updated the model should not contain redundant semantics (related to conciseness).

The characteristics of Req 4 are also used for software requirements specifications [1]:
- *Unambiguous*. The definition should only allow a single interpretation
- *Verifiable*. The information can be checked for correctness.
- *Traceable*. The origin of the definition can be determined

The characteristics of Req 5 and Req 6 are described in ISO/IEC 9126 [25], as software product quality (sub)characteristics. Usability and maintainability should be checked in the validation and deployment phases.
- *Maintainability*. The capability of the product to be modified
- *Usability*. The capability of the product to be understood, learned, used and liked by the user, when used under specified conditions

3.2.2 Design

As described above, the CF can be viewed as domain *ontology*:

An ontology is a definition of common concepts and relationships used to describe and represent an area of knowledge (i.e. a specification of a conceptualization [23]).

In Noy et al. [37], several guidelines are given for ontology development. We will apply ontology engineering as used in the development of the Common Warehouse Metamodel (CWM) [13]. The metamodel is described in the Unified Modelling Language (UML).
In the CWM Business Nomenclature (see UML Class Diagram in Figure 2) two levels are distinguished:

- a *taxonomy* with *concepts* at semantic level (conceptual model or domain model),
- a *glossary* with *terms* at representation level (terminology).

A concept can be related to other concepts and is identified by a number of terms. A term can be related to other terms and can be used in the description of concepts. A term is described in its definition.

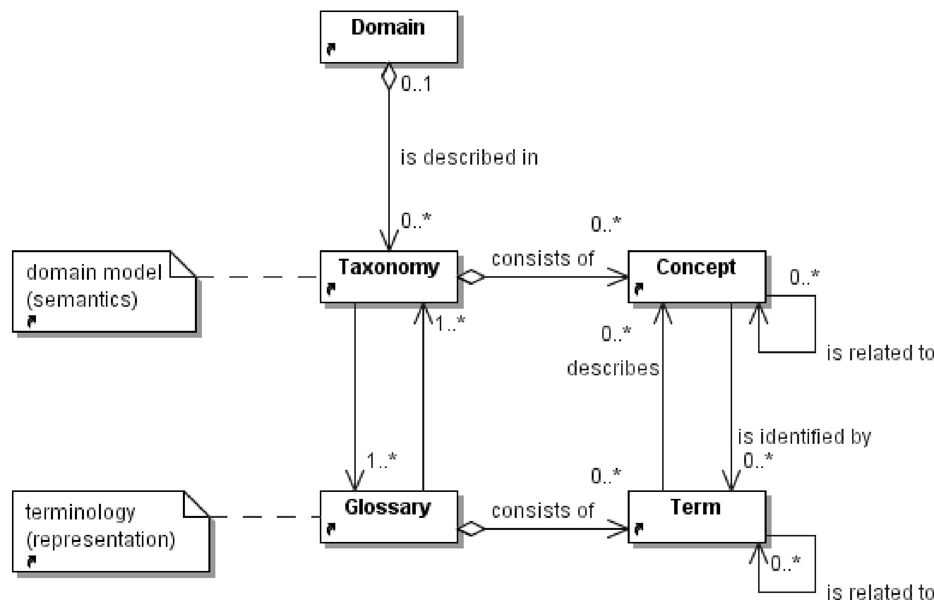


Figure 2. Relation between Taxonomy and Glossary (fragment of CWM [13])

Relations between terms
In CWM, the relation between terms in a glossary can be one of:
- *see also*: points to related term
- *antonym*: (or contrast), points to term with opposite meaning
- *synonym*: points to different term with same meaning
- *preferred*: points to preferred term
- *narrower*: points to more specialized term
- *wider*: points to more generalized term
- *acronym*: points to spelled-out term (and vice versa)

We add also the relationship:

- *homonym* (or multiple definition): points to term with same name but with different meaning. (These terms can be distinguished by a numerical postfix)

Similar relationships between terms have been used in other dictionaries (e.g. [19]).

Relations between concepts

The relation between concepts in a taxonomy can be generalization/specialization, aggregation and composition, association and dependency, where needed enriched with navigation direction, labels and multiplicities.

Definitions

There are many types of definitions, such as denotative definitions, connotative definitions and operational definitions [12].

- *Denotative definitions* rely on techniques that identify the extension(s) of the general term being defined.
- In *connotative definitions*, the intension of a term consists of the attributes shared by all the objects denoted by the term, and shared only by those objects.
- An *operational definition* of a term states that the term is correctly applied to a given case if and only if the performance of specified operations in that case yields a specified result.
- In a *precizing definition,* the vagueness of a term is reduced by restricting the meaning in a particular context.

We recommend using preferably denotative definitions (also called the genus-difference definition) with the structure:

<Concept> is <more general concept> with <specific properties>.

Copi and Cohen [12] provide some *guidelines* for this type of definitions:

- Focus on essential properties
- Avoid circularity
- Capture correct properties (not too broad, not too narrow)
- Avoid ambiguous and figurative language; be factual, not persuasive.
- Be affirmative rather than negative

The ontology design described above conforms to the objectives of the common foundation task. It is one of the critical success factors identified for the AOSD NoE (see section 2.1): *"Sharing a common terminology and associated conceptual model is a key contributing factor to effective technical discussions."*

3.2.3 Implementation

In this section we describe how the CF can be implemented, with modeling language and tools could be used. To support modification requests (MR) on the ontology we may use a web-based modification request tracking tool such as Bugzilla [8]. An important feature of these tools is the email notification.

We selected Bugzero [7] as modification tracking tool. BugZero has an adaptable workflow of modification requests, a customizable modification request form and access control for

different groups of users. A simple Workflow for Modification Requests is used (see Figure 3).

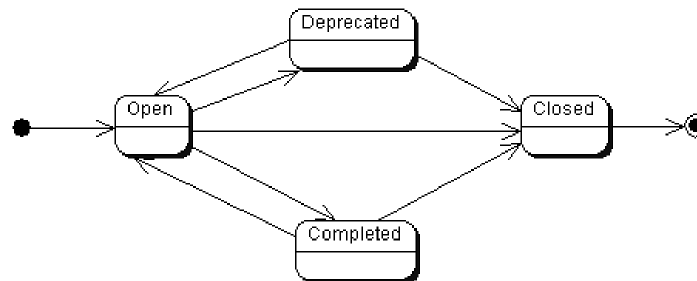


Figure 3. Simplified Workflow of Modification Requests

The states of a Modification Request are the following:

- *Open*: definition can be modified
- *Completed*: definition is stable
- *Deprecated*: term is not in use anymore
- *Closed*: further modification is not allowed

3.3. Collaborative Approach

Holsapple [24] describes a number of approaches to ontology design: inspiration, induction, deduction, synthesis and collaboration. The collaboation approach is described as follows:

"With a collaborative approach to ontology design, development is a joint effort reflecting experiences and viewpoints of persons who intentionally cooperate to produce it. Chances for relatively wide acceptance are enhanced if these persons are diverse in the contributions they make. This helps reduce blind spots in the ontology and enrich its content. On the other hand, coordination of the design process may suffer if too many persons are directly involved. The process itself could range from being strongly anchored, with a proposed ontology as a starting point for iterative improvements, to comparatively unstructured serendipitous discussion. In order to execute a collaborative approach, a consensus-building mechanism needs to be employed."

The development process for the collaborative approach is depicted in Figure 4. The four phases in the process are: preparation, anchoring, iterative improvement and application.

3.3.1 Procedure

We briefly describe the phases in the collaborative approach (see Figure 4):

1. **Preparation**.
   In this phase, the design criteria for the ontology are defined, as well as the evaluation criteria, and possible boundary conditions
2. **Anchoring**.
   In this phase, an intial ontology is established

3. **Iterative Improvement**.
   In this phase, reviews are collected from other participants. The ontology is revised untill consensus is reached.
4. **Application**.
   In this phase, the usability of the ontology is demonstrated in application domains.

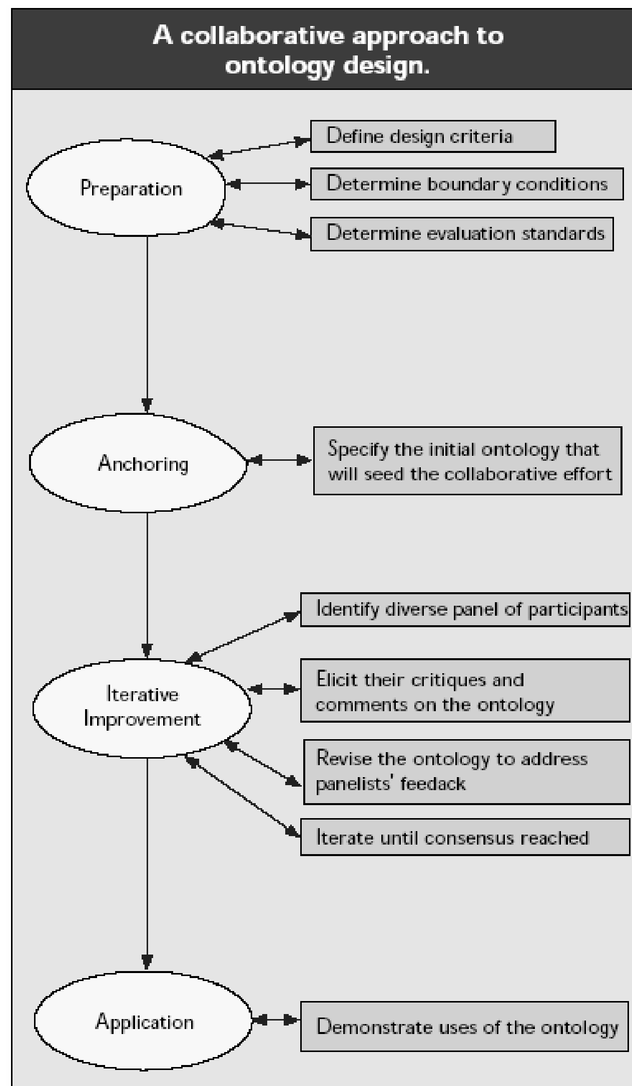


Figure 4. The development process in the collaborative approach [24]

We use a collaborative approach to ontology design, and describe activities in this process. We follow the phases in this collaborative approach for the development of the CF AOSD.

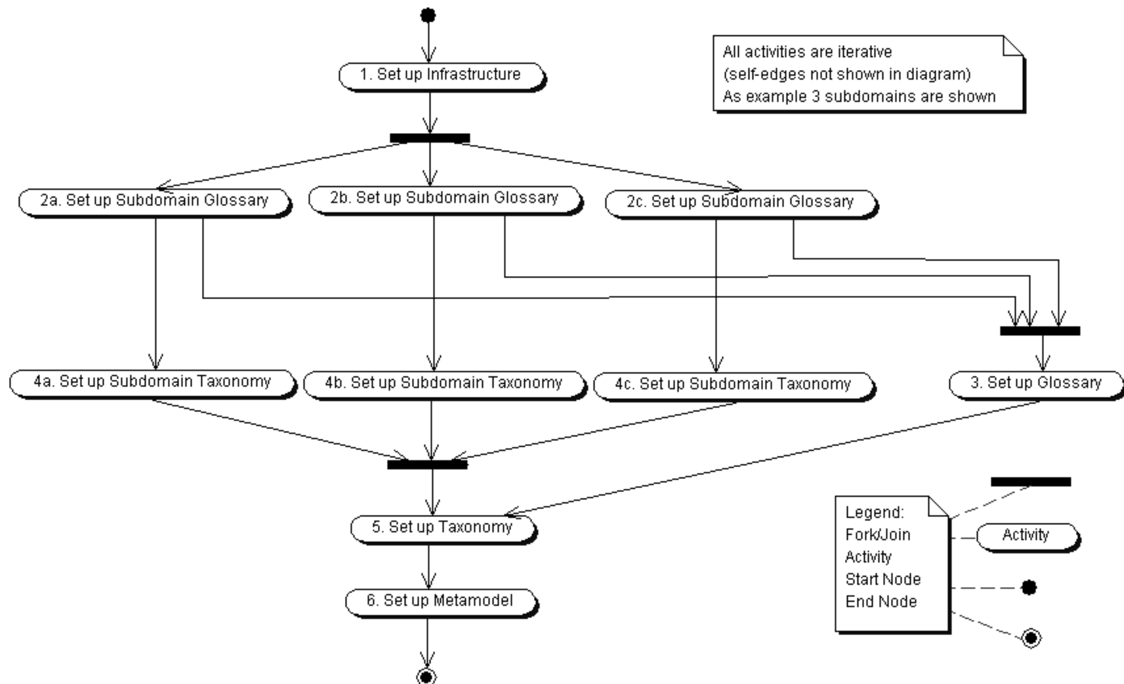


Figure 5. Activities in the development of the Common Foundation

3.3.2 Activities in the development of the Common Foundation

Based on the collaborative approach to ontology development, we carried out the following activities in the Common Foundation task (see Figure 5):

- A pilot project has been carried out for setting up the Infrastructure (activity 1) and an intial version of the AspectJ Glossary (activity 2a).
- A first version of the AspectJ Taxonomy has been set up (activity 4a), which also resulted in an updated version of the AspectJ Glossary.
- An initial version of the ComposeStar Glossary has been set up (activity 2b) and the ComposeStar Taxonomy (activity 4b).
- A glossary for aspect-oriented terms and concepts in the software lifecycle development phases requirements analysis, architectural design and detailed design has been set up (activity 3c).
- A common glossary has been set up (activity 3)
- A common taxonomy has been set up (activity 5)

Setting up a metamodel (activity 6) is not part of this work package but will be pursued in the Formal Methods Lab. It will build upon the achievements in the Common Foundation task.

3.4. Infrastructure

In this section, we describe the infrastructure (activity 1) for the ontology development with an extension to our ontology model.

3.4.1 Tool Support

There are several dedicated ontology languages such as OWL [38]. We decided to use class diagrams in the Unified Modelling Language (UML [47]) to represent conceptual domain models.

Based on a pilot study, the following infrastructure is set up for support of the ontology development (see Figure 6):

- WEBsina Bugzero Version 3.9.7 (multi-user) - Modification Request System [7]
- MicroSoft Access 2003 - Database [36]
- Apache Tomcat 4.1 - Web Server [4]
- Borland Together 2005 - UML Tool [5] [6]

The terms in Bugzero are transferred manually to the UML Tool including a hyperlink to the ID of the term. The UML tool generates Html with class diagrams including links to the terms in Bugzero. These documents can be accessed by any browser.

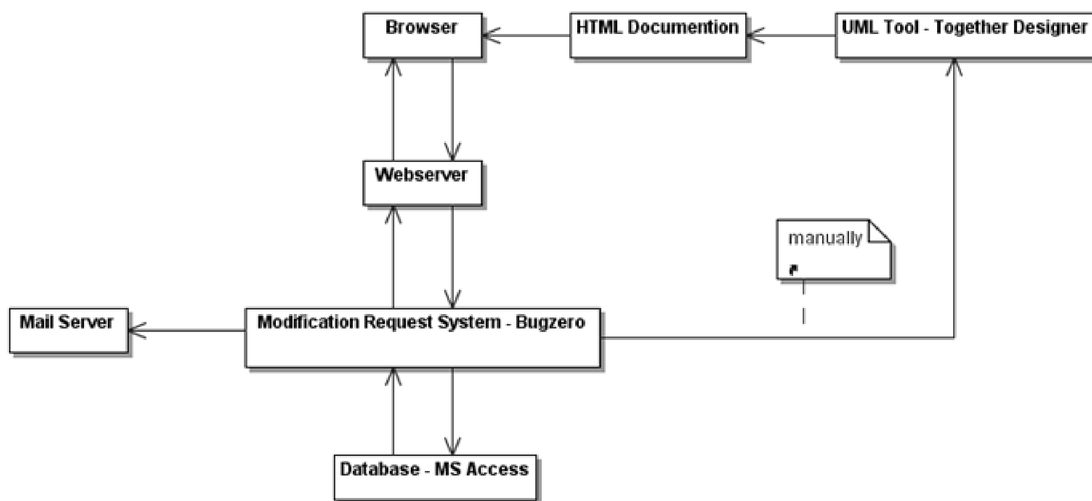


Figure 6. Tool Support for Collaborative Ontology Development

3.4.2 Model, View and Conceptual Framework

The taxonomy (domain model) contains many concepts. In order to cope with the complexity of the model we use views of the model.

- *A View is a model which is completely derived from another model (the base model). A view cannot be modified separately from the model from which it is derived. Changes to the base model cause corresponding changes to the view. (Gardner et al., 2004 [22])*

A similar distinction is made in the UML between model and diagram. (UML Reference Manual [41])

- *A Model is a semantically complete abstraction of a system - from a particular view-point - consisting of model elements. A Model element is an abstraction drawn from the system being modelled.*
- *A Diagram is a graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements).*

In this report, the domain model (or taxonomy) is an abstraction of the domain, i.e. the knowledge. Moreover, we use *view* and *concept diagram* interchangeable.

A concept related to *View* is a *Conceptual Framework.*
- *A Conceptual Framework is a coherent collection of related concepts*

We use this concept in Chapter 7.

In Figure 7 we show the concepts used for our ontology development including categories of the glossary and views of the taxonomy. For convenience, we assume a one-to-one mapping between Categories and Subdomains.

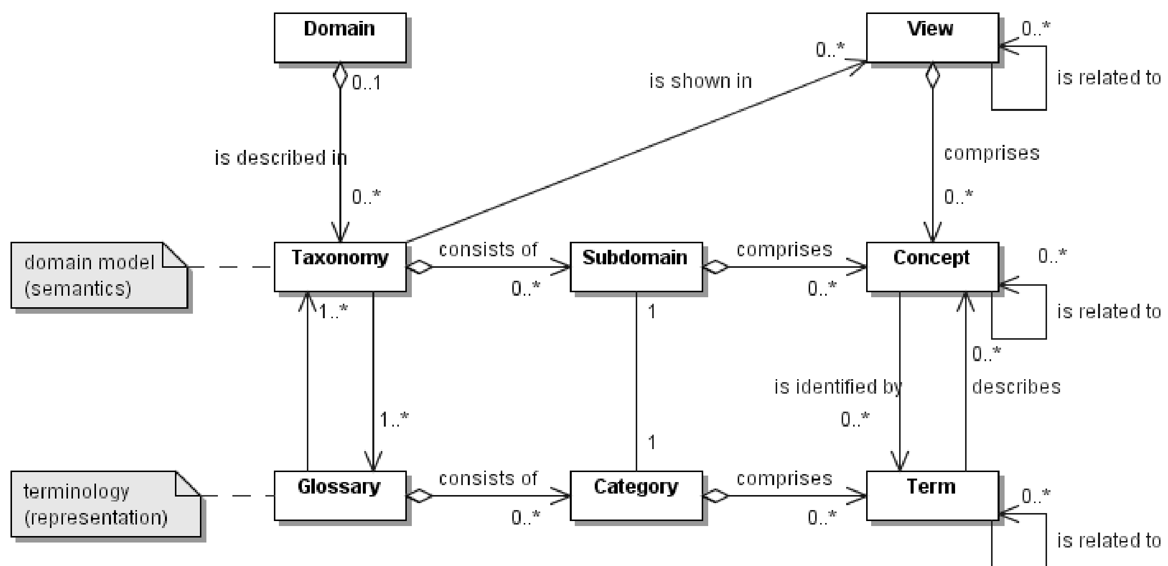


Figure 7. Ontology Concepts with Views (modified version of Figure 2)

The lower part of this figure (Glossaries, Categories and Terms) is covered by the Modification Request System Bugzero. The upper part is covered by the UML Tool Together Designer. All concepts in the UML models have a hyperlink to their corresponding definition in Bugzero. We use the stereotype <<view>> to distinguish views from concepts.

3.4.3 Modelling Taxonomies

Taxonomies and Subdomains are modelled in UML Packages. Concepts and Views are modelled in UML Classes. We follow the following conventions (see Figure 8).

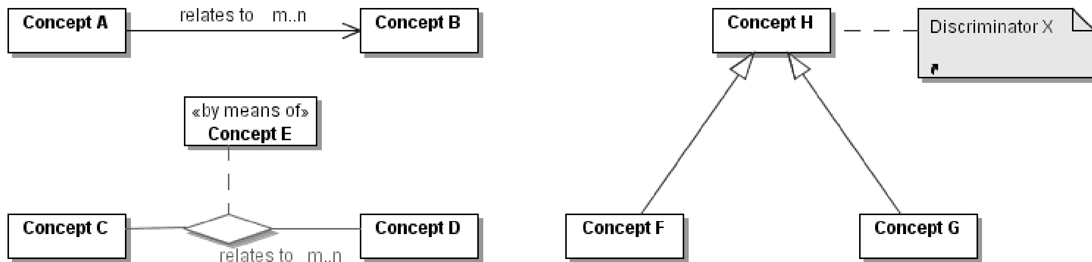


Figure 8. Model conventions for associated and specialized concepts

Roles are given in verb phrases (following the convention given by Mellor et al. [34]). The following description can be derived from the diagrams. E.g.

Concept A relates to m..n Concepts B

Relations can be bidirectional: in that case the roles and multiplicities are given for both association ends. The descriptions can be derived in both directions.

In some cases, the relation is enriched with more information modelled as an association class. In that case, the following description can be derived from the diagrams. E.g.

Concept C relates to m..n Concepts D by means of Concept E

Usually, concepts in association classes are described in operational definitions.

In case of specialization of concepts, e.g. Concepts F and G are specialized from Concept H, we describe the discriminator for the generalization/specialization relation.

Concept F is a (special) Concept H for which X has value x

This corresponds to a denotative definition of the concept.

These conventions are used in the taxonomy as described in the following sections.

3.5. Conclusion

In this chapter, we described a disciplined process to the development of the common foundation. This ontology engineering process is similar to the development process of software, with requirements, a design and implementation phase of the ontology. We described the requirements for the Common Foundation, among others: comprehensive, general acceptable, consistent, unambiguous and accurate. For the design, we selected the Common Warehouse Metamodel, with a distinction between a glossary with the definition of terms (terminology), and a taxonomy with concepts and relations between concepts (conceptual domain model). We selected a collaborative approach to ontology development. We described web-based support for the development of the glossaries. Furthermore, we introduced the terminology for ontology's that is used in the other parts of this report. This terminology is summarized in Appendix A.

4. AspectJ Terms and Concepts

In this chapter, we describe terminology and a taxonomy for AspectJ. We selected AspectJ as a major representative of aspect languages. We used AspectJ to set up the infrastructure for ontology development and to examine representations for taxonomies. The goal of the terminology and taxonomy of AspectJ is to use this as a base for the generalization of terms and concepts within the Common Foundation.

4.1. Introduction

In chapter 3, we described the relation between the glossary (terminology) and taxonomy (conceptual domain model) in an ontology. In this chapter we describe both a glossary of AspectJ terms and a taxonomy of AspectJ concepts.
In chapter 3, we also described the procedure in the Collaborative Approach to ontology development. For the AspectJ glossary and taxonomy, we followed the following procedure:

- *Preparation*: The boundary condition is that this glossary and taxonomy on AspectJ should support the establishment of the common glossary and taxonomy. It is not the goal of the Common Foundation task to establish the ultimate AspectJ ontology.
- *Anchoring*: We extracted the initial set of terms from the book AspectJ in Action by Ladded, 2003 [28]. The terms were recorded in the Modification Request System (see section 3.4).
- *Iterative improvement*: The initial set of terms was reviewed by participants of the AspectJ pilot study. Changes in the definitions of terms were recorded in the Modification Request System.
- *Application*: The insight gained by setting up the AspectJ glossary and taxonomy are used in other domains (e.g. ComposeStar, see Chapter 5) and for setting up the common terminology and common taxonomy (Chapter 6 and 7 respectively).

Overview of chapter

We describe the glossary of AspectJ terms (section 4.2), followed by an outline of the AspectJ taxonomy using different views on this taxonomy (section 4.3). The complete glossary is given in Appendix C.

4.2. AspectJ Glossary

The initial glossary contained 70 terms and definitions mainly based on the book by Laddad (2003) [28]. After modification by the experts the glossary contained 90 terms and definitions. In the current state there are 123 terms (see Table 2). In Appendix C an overview of the terms is given in a UML package diagram.

In Appendix C an alphabetical list is given of AspectJ terms in the four categories with their defintions. In the initial glossary, the categories Basic Concepts, Advanced Concepts

and References were included. During the pilot study there was a need for another category for terms not specific to AspectJ. This category was named General AOSD Concepts.

	Glossary At Milestone M 1.2 30-11-2004	Glossary At Milestone M 1.3 22-02-2005
Category	**Number of terms**	**Number of terms**
Basic terms	20	23
Advanced terms	41	66
General AOSD terms	25	28
References	4	6
Total	90	123

Table 2 Summary statistics of AspectJ Glossary

4.3. AspectJ Taxonomy

In this section we first refine the taxonomy by describing views on the AspectJ domain. Then we show a number of views of the AspectJ taxonomy.

4.3.1 Views of AspectJ Taxonomy

In the current version of the AspectJ taxonomy we distinguish 15 views. These views (see Section 3.4.2) are just for convenience (*divide and conquer*). An overview of these views is given in a View Model (see Figure 9). We use the stereotype <<view>> to distinguish views from concepts.

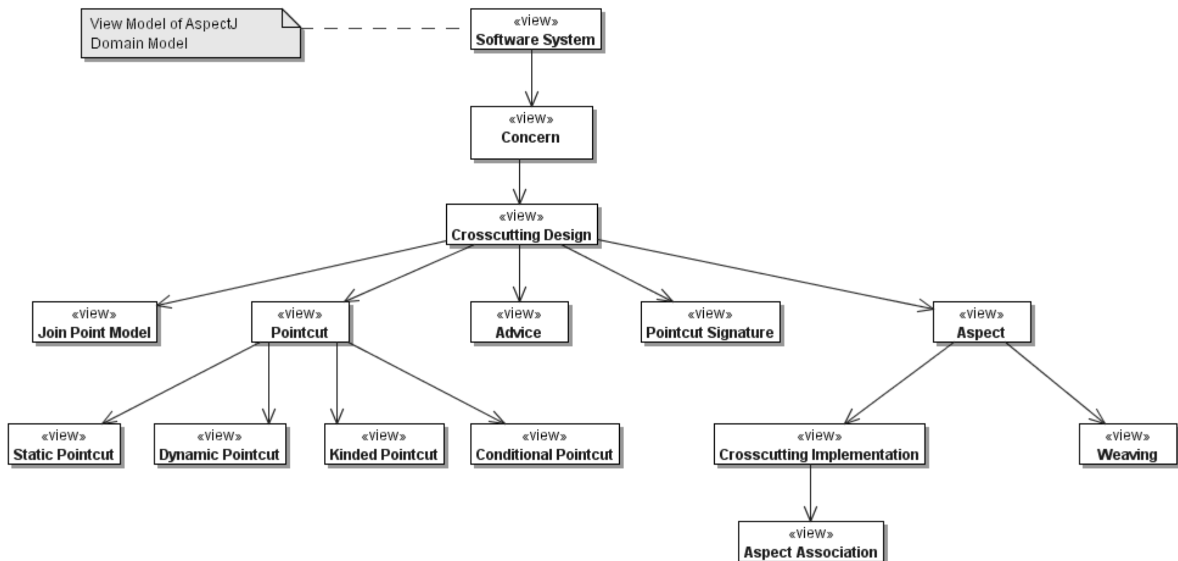


Figure 9. Model of Views of the AspectJ Taxonomy

The views are included in the Appendix. In the generated HTML-documentation there are hyperlinks to the corresponding diagrams. Here, we discuss a few example views. We start with the Software System.

4.3.2 Software System View

In this section we discuss the view of the AspectJ taxonomy as seen from the Software System with related concepts, especially the concept Concern. The Software System View is shown in Figure 10.

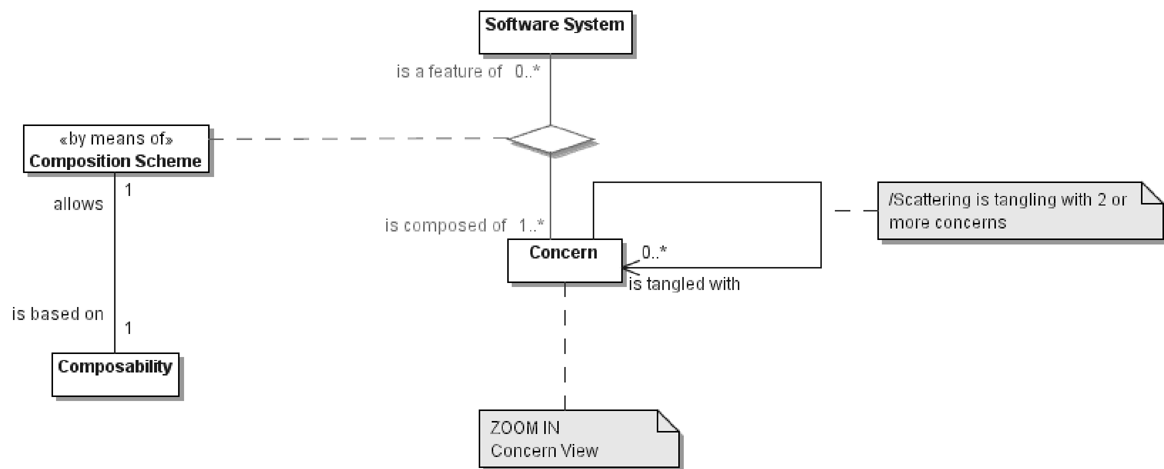


Figure 10. Software System View of the AspectJ Taxonomy

There is an association between the concepts Software System and Concern with an association class Composition Scheme. The description of this association is as follows:
- *A Software System is composed of 1..* Concerns by means of a Composition Scheme.*
- *A Concern is a feature of 0..* Software Systems by means of a Composition Scheme.*

The other associations are described as follows:
- *A Concern is tangled with 0..* Concerns*
- *A Composition Scheme is based on Composability (of Concerns)*
- *Composability allows a Composition Scheme.*

All concepts have a hyperlink to their corresponding definition in Bugzero. From this view we may zoom in on the Concern View using a hyperlink in the note.

4.3.3 Concern View

In this section, we discuss the view of the AspectJ taxonomy as seen from the concept Concern with related concepts, especially the concept Aspect. The Concern View is shown in Figure 11.

In the Concern View we see a specialization of the concept Concern into two concepts Core Concern and Crosscutting Concern. The discriminator for this specialization is the Aspectual Decomposition. The association is described as follows:

- *A Core Concern is crosscut by 0..* Crosscutting Concerns by means of Crosscutting (design level)*
- *A Crosscutting Concern crosses over 0..* Core Concerns by means of Crosscutting (design level)*

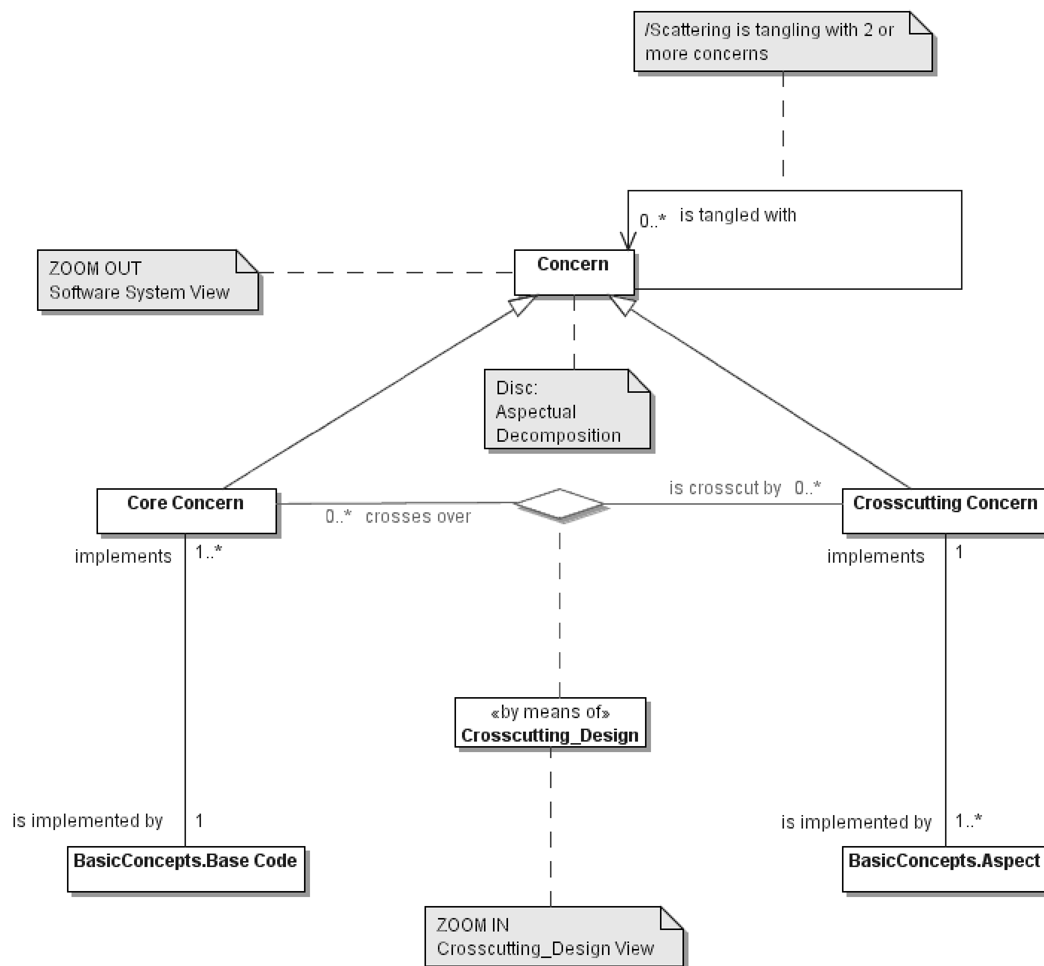


Figure 11. Concern View of the AspectJ Taxonomy

4.3.4 Crosscutting Design View

In this section we discuss the view of the AspectJ taxonomy as seen from the concept Crosscutting at design level with related concepts. The Crosscutting Design View is shown in Figure 12.

From the concept Aspect, two crosscutting structures are distinguished: static and dynamic crosscutting. Static crosscutting is defined with Inter-type declarations. For dynamic crosscutting we rely on Advices and Pointcuts.

From here one may zoom in on several other views such as Advice View, Join Point Model View and Pointcut View.

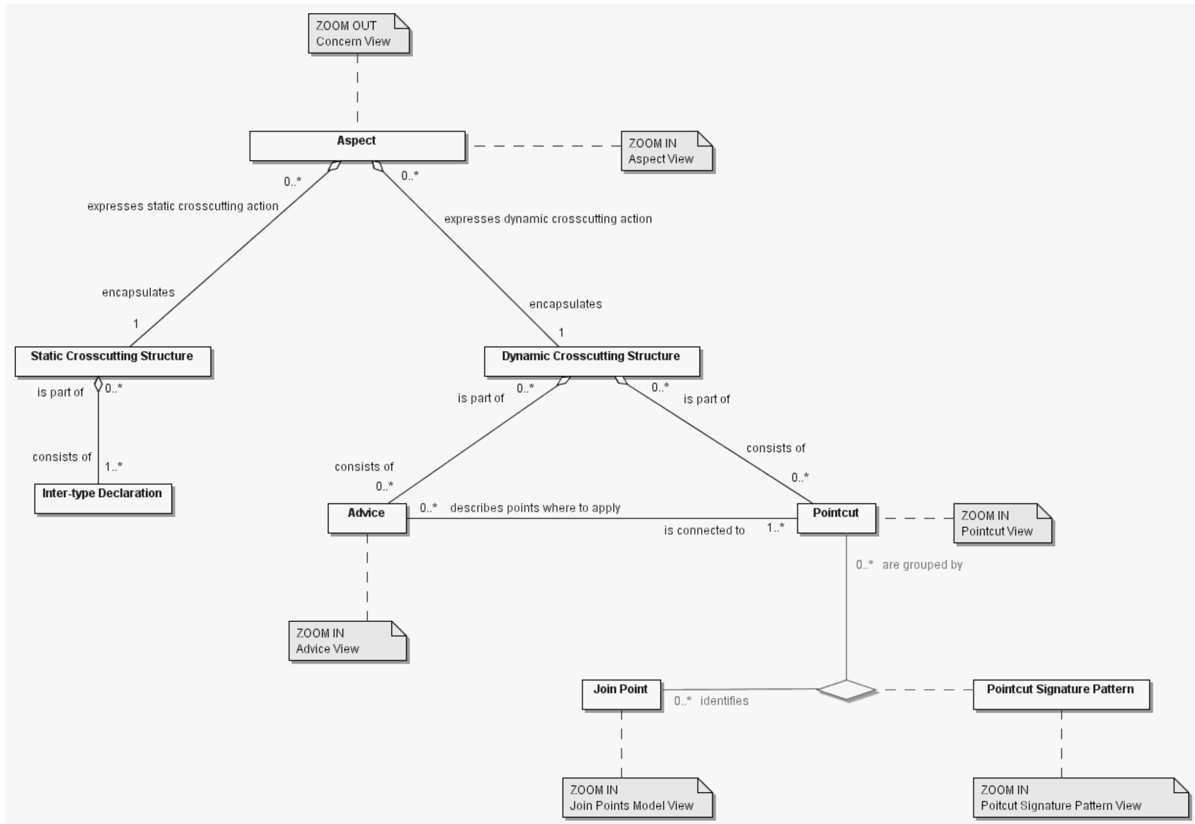


Figure 12. Crosscutting Design View of the AspectJ Taxonomy

4.3.5 Aspect View

In this section we discuss the view of the AspectJ taxonomy as seen from the concept Aspect with related concepts. The Aspect View is shown in Figure 13.

The concept Base Code is related to Aspect based on Crosscutting at implementation level.
The concept Aspect has two reflexive relations based on Aspect Precedence and Aspect Inheritance respectively.
The concept Aspect can be specialized to Concrete Aspects and Abstract Aspects based on the discriminator Abstractness.

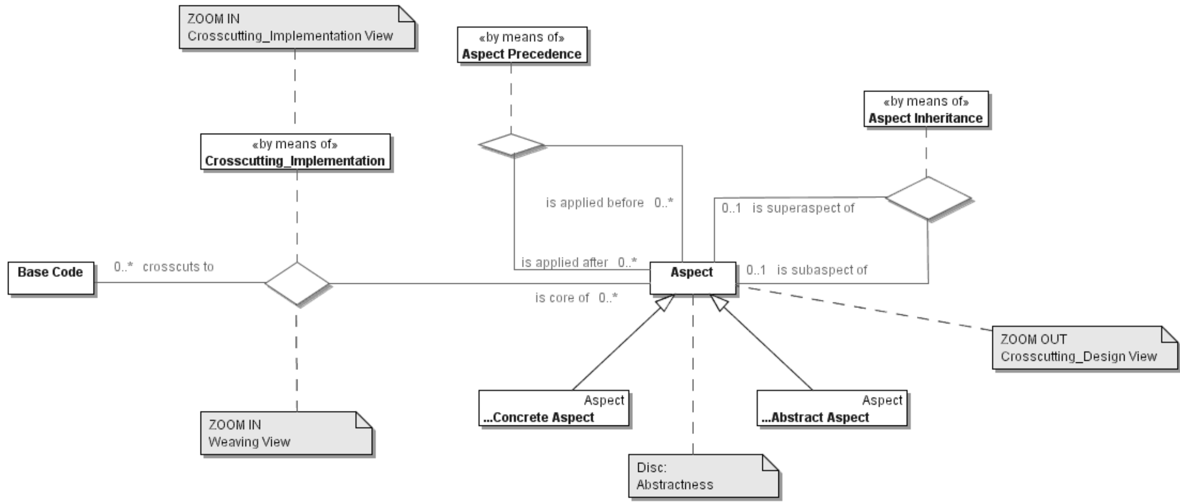


Figure 13. Aspect View of the AspectJ Taxonomy

4.3.6 Pointcut Signature View

In this section we discuss the view of the AspectJ taxonomy as seen from the concept Pointcut Signature Pattern with related concepts. This view is shown in Figure 14.

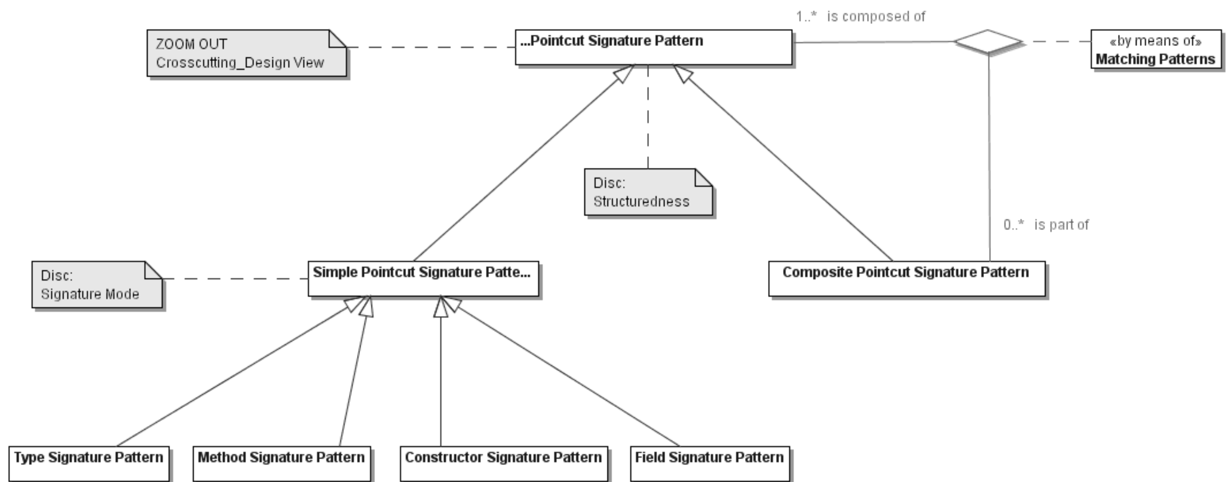


Figure 14. Pointcut Signature View of the AspectJ Taxonomy

In this view we recognize the Composite Design Pattern [21] for the concepts Pointcut Signature Pattern, Composite Pointcut Signature Pattern and the concept Simple Pointcut Signature Pattern with its specializations.

4.3.7 Other Views

As we have seen in the View Model (Figure 9) there are many other views. They will not be discussed here in detail. They are enumerated alphabetically in Appendix E.

4.3.8 Analysis of AspectJ Views

In this section we analyse how concepts are distributed over the views. Here we summarize the results (see Table 3).

View	**Number of Concepts in View**
Advice View	8
Aspect Association View	6
Aspect View	10
Concern View	10
Conditional Pointcut View	5
Crosscutting Design View	12
Crosscutting Implementation View	8
Dynamic Pointcut View	7
Join Point Model View	21
Kinded Pointcut View	17
Pointcut Signature View	8
Pointcut View	13
Software System View	6
Static Pointcut View	11
Weaving View	12

Table 3. Number of Concepts per View in AspectJ Domain

The Join Point Model View and the Kinded Pointcut View (see Appendix E) comprise a large number of concepts because of the many specialized concepts.

We found that some terms from the AspectJ Glossary did not appear at all in one of the views. Some of these terms are synonyms or terms similar to other terms. Moreover we introduced new terms needed to cover the concepts used in the views. These terms are added to the Glossary in Bugzero.

Similar terms are the following:

- Aspect Association - Aspect Instantiation
- Weaving Time - Weaving Mode
- Introduction - Static Crosscutting
- Compile Time Declaration - Compile Time Error and Warning Declaration
- Aspectual Recomposition - Weaving
- Dynamic Crosscutting - Aspectual Decomposition

Similarities and differences between these terms will be established in the review process.

New terms are the following:

After Returning, After Throwing, Composite Pointcut, Concrete Pointcut, Constructor Signature Pattern, Dynamic Crosscutting Structure, Dynamic Weaving, Field Signature Pattern, Initialization Join Point, Initialization Pointcut, Invocation Join Point, Invocation Pointcut, Matching Patterns, Method Signature Pattern, Normal After, Primitive Pointcut, Property-based Pointcuts, Simple Signature Pattern, Static Crosscutting Structure, Static Weaving, Type Signature Pattern.

The next step will be a systematic review of the definitions derived from the taxonomy and make them consistent with the terms defined in the glossary. In the review we will compare our taxonomy with other approaches found in the literature (e.g. [9], [11]).

4.4. Conclusion

In this chapter, we described terminology and a taxonomy for AspectJ. We selected AspectJ as a major representative of aspect languages. In a Pilot study, we used AspectJ to set up the infrastructure for ontology development and to examine representations for taxonomies. The goal of the terminology and taxonomy of AspectJ is to use this as a base for the generalization of terms and concepts within the Common Foundation. We started with a large initial glossary with 70 terms based on the book by Ladded, 2003 [28]. It appeared difficult to cope with so many definitions at the same time. It was decided to split the glossary in three categories: basic terms, advanced terms and general AOSD terms. Changes in definitions were tracked in the Modification Request System. Most changes were recorded on the definitions of Aspect, Advice, Concern, Crosscutting Concern, and Join Point. The current AspectJ glossary contains more than 100 terms and definitions. For the taxonomy, we used views to show how important concepts in AspectJ are related with other concepts. The definitions in the glossary and conceptual model in the taxonomy should evolve further during the AOSD-Europe project. The glossary with AspectJ definitions is summarized in Appendix C.

5. ComposeStar Terms and Concepts

In this chapter, we describe terminology and a taxonomy for ComposeStar. We selected ComposeStar to represent another approach in aspect languages than AspectJ. The goal of the terminology and taxonomy of ComposeStar is to use this, besides the terminology and taxonomy for AspectJ, as a base for the generalization of terms and concepts within the Common Foundation.

5.1. Introduction

In chapter 3, we described the relation between the glossary (terminology) and taxonomy (conceptual domain model) in an ontology. In this chapter we describe both a glossary of ComposeStar terms and a taxonomy of ComposeStar concepts.

The ComposeStar ontology is developed along side the AspectJ ontology in order to be able to generalize definitions of terms from both domains for the common ontology.

In chapter 3, we also described the procedure in the Collaborative Approach to ontology development. For the ComposeStar glossary and taxonomy, we followed the following procedure:

- *Preparation*: The boundary condition is that this glossary and taxonomy on ComposeStar should support the establishment of the common glossary and taxonomy. It is not the goal of the Common Foundation task to establish the ultimate ComposeStar ontology.
- *Anchoring*: The initial set of terms for ComposeStar was established in a number of meetings of the ComposeStar team at the University of Twente and recorded in the Modification Request System (see section 3.4).
- *Iterative improvement*: The initial set of terms was reviewed by members of the ComposeStar team. Changes in the definitions of terms were recorded in the Modification Request System.
- *Application*: The insight gained by setting up the ComposeStar glossary and taxonomy and the AspectJ glossary and taxonomy are used for setting up the common terminology and common taxonomy (Chapter 6 and 7 respectively).

Overview of chapter

We describe the glossary of ComposeStar terms (section 5.2), followed by an outline of the ComposeStar taxonomy using different views on this taxonomy (section 5.3). The complete glossary is given in Appendix D.

5.2. ComposeStar Glossary

As outlined in the introduction (section 2) we will set up glossaries and taxonomies for a number of domains. Parallel to the AspectJ domain we started working at the ComposeStar domain, related to the Composition Filters approach at the University of Twente [10]. In Figure 15 we give an overview of terms defined in the Glossary for ComposeStar.

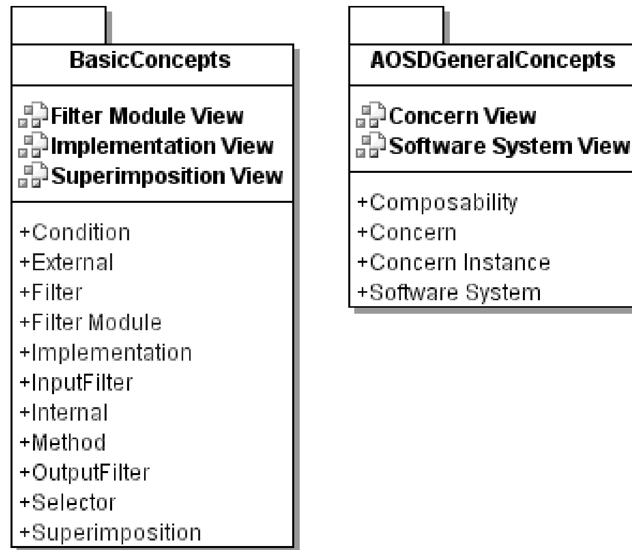


Figure 15. ComposeStar Terms per Category

In the Table 4 we summarize the number of terms per category for this initial version of the ComposeStar Glossary.

	Glossary At Milestone M 1.3 22-02-2005
Category	**Number of terms**
Basic terms	11
Advanced terms	0
General AOSD terms	4
References	2
Total	17

Table 4 Summary statistics of ComposeStar Glossary

It is clear that this glossary is just a start. A group of researchers at the University of Twente elaborate the definitions in this glossary.

5.3. ComposeStar Taxonomy

In this section we show concepts in a number of views of the ComposeStar taxonomy.

5.3.1 Views of ComposeStar Taxonomy

In the current version of the ComposeStar taxonomy we distinguish 5 views. An overview of these views is given in a View Model (see Figure 16).

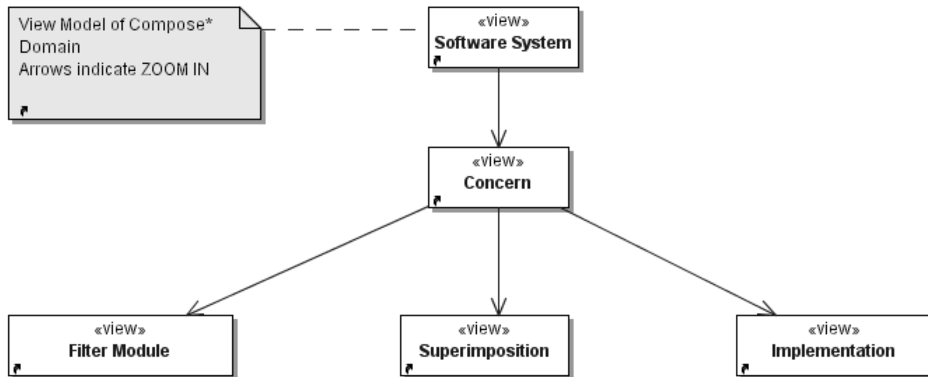


Figure 16. View Model of ComposeStar Domain

As in AspectJ we first encounter the Software System View and Concern View. Filter Model and Superimposition are specific for the ComposeStar domain.

5.3.2 Concern View

In this section we show the view of the ComposeStar taxonomy as seen from the concept Concern with related concepts, especially the concept Superimposition and Filter Module (see Figure 17).

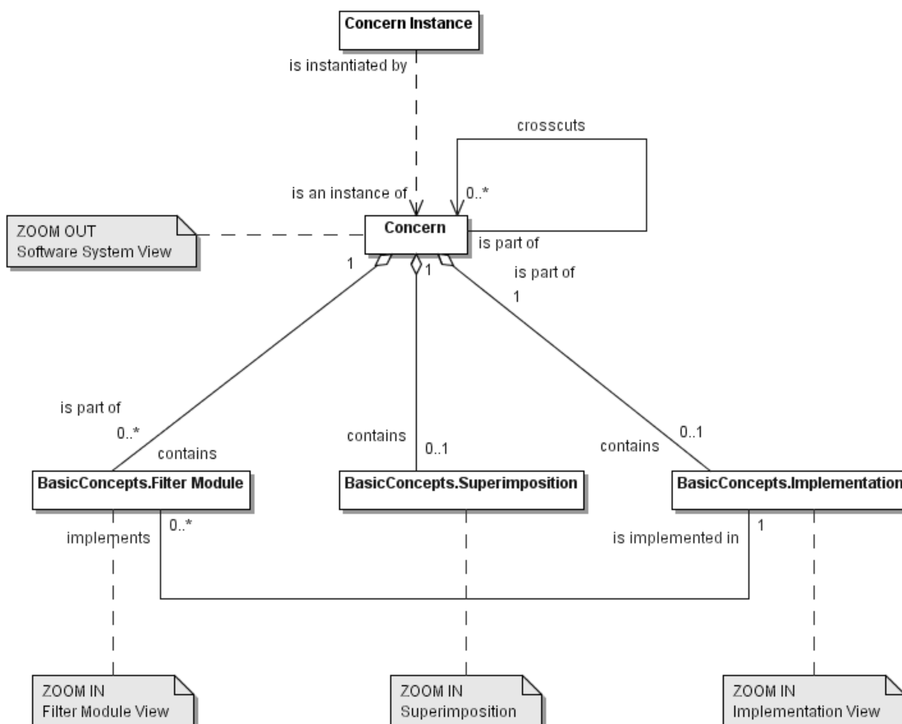


Figure 17. Concern View of the ComposeStar Taxonomy

5.3.3 Software System View

In this section we show the view of the ComposeStar taxonomy as seen from the Software System (see Figure 18).

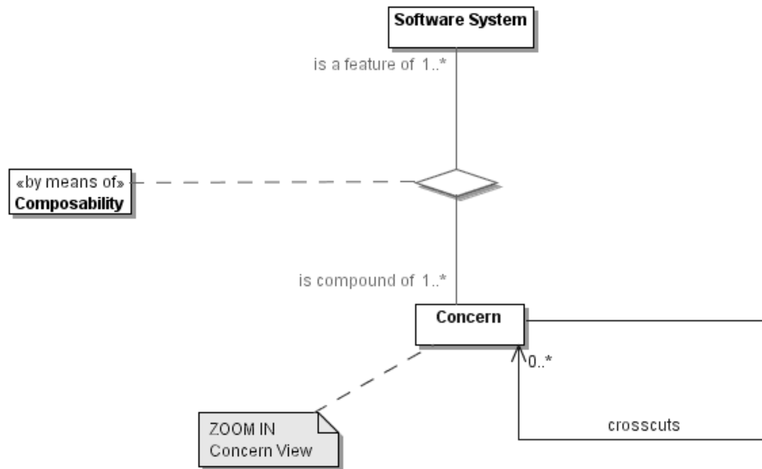


Figure 18. Software System View of the ComposeStar Taxonomy

5.3.4 Superimposition View

In this section we show the view of the ComposeStar taxonomy as seen from the concept Superimposition with related concepts (see Figure 19).

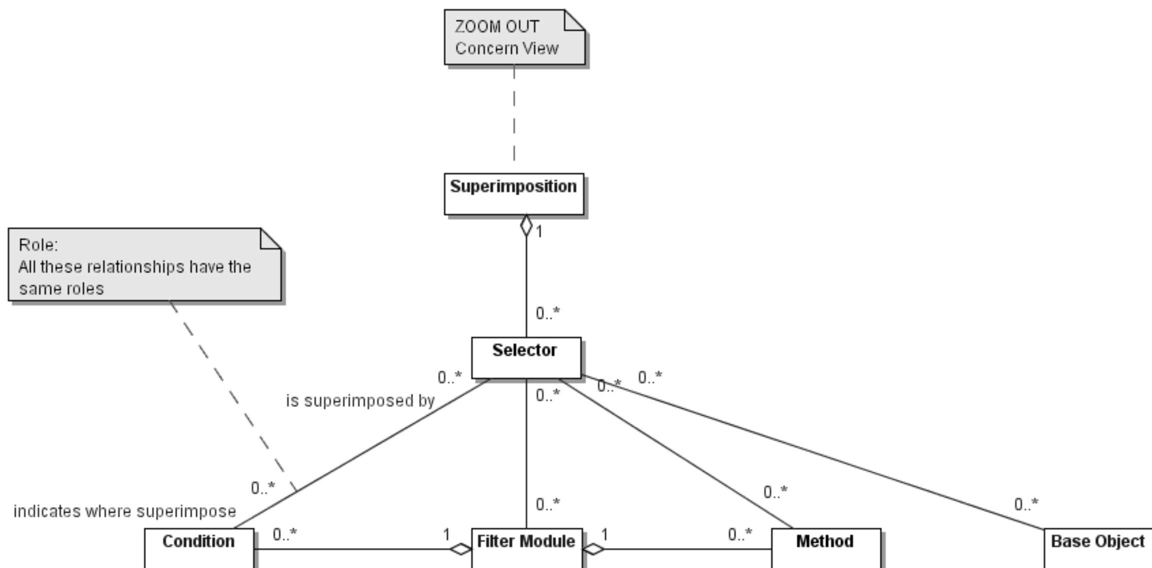


Figure 19. Superimposition View of the ComposeStar Taxonomy

5.4. Conclusion

In this chapter, we described a glossary and taxonomy for ComposeStar. We selected ComposeStar to represent another approach in aspect languages than AspectJ, namely the composition filters approach. The goal of the terminology and taxonomy of ComposeStar is to use this, besides the terminology and taxonomy for AspectJ, as a base for the generalization of terms and concepts within the Common Foundation. The current ComposeStar glossary contains only 11 basic terms and 4 general AOSD terms. During the discussions in the ComposeStar project, it appeared difficult to capture the state-of-the-art of terminology on composition filters, and not to mix it with new research developments, and not to mix it with commonly used AspectJ terminology. It appeared to be a very useful activity for the ComposeStar project to set up such a glossary. The definitions and conceptual models will be developed further as part of the ComposeStar project. The glossary with definitions of the ComposeStar terms is given in Appendix D.

6. Development of Common AOSD Terminology

In this chapter, we describe the set up of a glossary of common AOSD terms. It is important to realize that this terminology reflects the current state-of-the-art, although not everybody will fully agree on all definitions. One could say that the definitions in this chapter bear a time stamp: they are not the final verdict on AOSD terminology. The glossary with definitions should evolve during the AOSD-Europe project according to new developments and new insights. Eventually, the terminology in the glossary (this chapter) and the conceptual framework (subsequent chapter) should be consistent which each other. This consistency should evolve during the project.

6.1. Introduction

In chapter 3, we described the relation between the glossary (terminology) and taxonomy (conceptual domain model) in an ontology. In this chapter, we describe a glossary of common AOSD terms.

The AOSD glossary is developed based on commonality analysis and generalization from the previous chapters on the AspectJ ontology (see Chapter 4) and the ComposeStar ontology (see Chapter 5), and on publications of among others Concern Modelling (Sutton et al. 2002, 2004 [44][45]), Aspectual Requirements Analysis (Rashid et al. 2003, [40]) and Aspectual Architectural Design (Tekinerdogan 2004, [46]).

In chapter 3, we described the procedure in the Collaborative Approach to ontology development. For the common AOSD glossary, we followed the following procedure:

- *Preparation*: The design criterion is that this common AOSD glossary generalizes from specific aspect languages, and from specific lifecycle phases in software development. The evaluation criterion is that the common AOSD should be - among others - general acceptable, consistent and accurate. These are the requirements explained in section 3.2.1.
- *Anchoring*: The initial set of terms and definitions was based on the book by Filman et al. (2005) [18]. This set was merged with a set of terms established at a Workshop on AOSD Terminology (University of Twente, Enschede). The terms and definitions were recorded in the Modification Request System (see section 3.4).
- *Iterative improvement*: The initial set of terms was reviewed by participants of the Workshop, and later by other participants from the AOSD-Project.
- *Application*: The insight gained by setting up the common AOSD glossary are used for setting up (part of) the common AOSD taxonomy (Chapter 7).

Overview of chapter

We describe the commonality analysis and the initial set of terms in the common AOSD glossary (section 6.2), followed by an outline of the definitions with alternatives (section 6.3). The common AOSD glossary is given in Appendix B.

6.2. Selection of core terms

There are many resources for setting up a glossary for AOSD. First of all, there are many publications (e.g. the AOSD bibliography by Filman [17], the Early Aspect website [15]). Moreover, there are several conference and workshop series such as the AOSD Conferences [2] and the FAOL Workshops [20]. There are numerous workshops on AOSD connected with other conferences. There are few attempts to set up common terminology for AOSD (e.g. Mehner & Rashid, 2003 [32][33]).

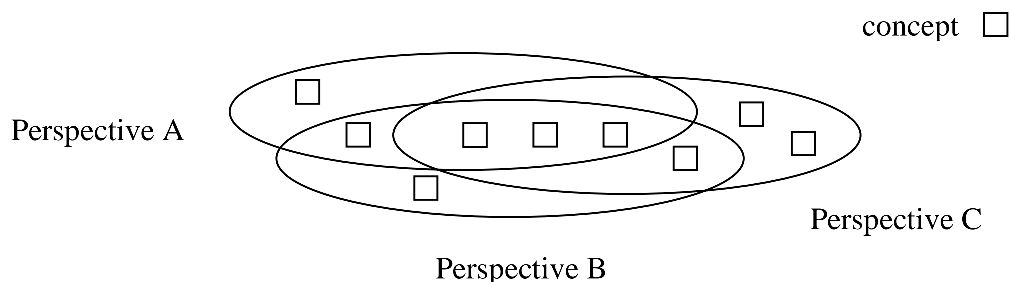


Figure 20. Common Terminology derived from different perspectives

There are many different perspectives (see Figure 20) from where AOSD terms are defined, such as aspect language implementations, aspectual requirements analysis, aspectual architectural design, concern modelling, formal aspect semantics.
Commonality analysis is a well-known technique in domain engineering (e.g. Czarnecki et al. 2000 [14]). A common glossary collects common terms and generalizes the definition such that the general definition could be used in the specific context. Generalization is not always possible because sometimes the definitions are too vague or are conflicting. An example of conflicting definitions for Join Point is given by Mechner & Rashid, 2003 [32].

Core terms

We selected an initial set of core terms for the common AOSD glossary. These terms are the following:

- Separation of Concerns, Tyranny of Dominant Decomposition, Composition, Weaving, Decomposition, Modularization
- Concern, Crosscutting Concern, Crosscutting, Scattering, Tangling
- Aspect, Advice, Pointcut, Join Point, Join Point Model

This initial set can be augmented in later phases of the AOSD-Europe project. We provided definitions for these terms and collected comments in several review rounds. The result is a list of terms with some alternative definitions. This set with preferred definitions is collected in Appendix B.

Criterions

The main criterions for selection definitions for the common AOSD glossary are (see also Chapter 3 for the other requirements):

- The definition should be *general acceptable*: this means that the definition is independent from specific lifecycle phases or specific aspect languages.
- The definition should be *consistent* with other definitions
- The definition should be *accurate* and *unambiguous*

Definitions with some alternatives are described in the following section. The discussion about the definitions are not finished.

6.3. Glossary

In this section, we give the definitions of the core terms in the AOSD-glossary with alternative definitions. The first definition is usually based on the Introduction (Chapter 1) in the book by Filman et al. (2005) [18].

Nr	**Term**	**Definition**	
1.	Separation of Concerns	Separation of concerns simplifies system development by allowing the development of specialized expertise and by producing an overall more comprehensible arrangement of elements. [18]	
2.		Separation of Concerns is an in depth study and realisation of concerns in isolation for the sake of their own consistency (adapted from “On the Role of Scientific Thought” by Dijkstra, EWD 447).	
3.		Separation of concerns addresses the issue of providing sufficient abstraction for each concern as a modular artefact.	
4.	Tyranny of Dominant Decomposition	No explicit definition in Filman et al. [18]	
5.		The Tyranny of the Dominant Decomposition refers to restrictions (or tyranny) imposed by the selected decomposition technique (i.e. the dominant decomposition) on software engineer's ability to modularly represent particular concerns.	
6.		The Tyranny of the Dominant Decomposition refers to restrictions imposed by this decomposition on the simultaneous use of other decompositions.	
7.	Composition	Composition is bringing together separately created software elements. [18]	
8.		Composition is the integration of multiple modular artefacts into a coherent whole.	
9.		Composition is the integration of artefacts into a whole. (Opposite of decomposition)	

10.	Weaving	Weaving is the process of composing core functionality modules with aspects, thereby yielding a working system. [18]	
11.		Weaving: Historically this term is used to refer to the composition of aspects with other concerns in the system. See composition.	
12.		Weaving is the composition of aspects with modules that represent other concerns in the system.	
13.	Decomposition	No explicit definition in Filman et al. [18]	
14.		Decomposition is the breaking down of a larger problem into a set of smaller problems which may be tackled individually.	
15.		Decomposition is the breaking down of a whole into smaller artefacts. (Divide and conquer. Opposite of composition)	
16.	Modularization	No explicit definition in Filman et al. [18]	
17.		Modularization is putting together (or partitioning) artefacts into entities called modules (usually aiming at low coupling and high cohesion).	
18.	Module	No explicit definition in Filman et al. [18]	
19.		Synonym: Modular Artefact	
20.		A module is an abstraction in the adopted language	
21.	Concern	A concern is a thing in an engineering process about which it cares. [18]	
22.		A concern is an interest which pertains to the system's development, its operation or any other matters that are critical or otherwise important to one or more stakeholders.	
23.	Crosscutting Concern	A crosscutting concern is a concern for which the implementation is scattered throughout the rest of an implementation. [18]	
24.		A crosscutting concern is a concern which cannot be modularly represented within the selected decomposition. Consequently the elements of crosscutting concerns are scattered and tangled within elements of other concerns.	
25.		A crosscutting concern is a concern, which is not modularly represented within the selected decomposition into modules, with as a result the occurrence of crosscutting.	
26.	Crosscutting	Crosscutting is a property of a concern for which the implementation is scattered throughout the rest of an implementation. [18]	

27.		Crosscutting is the scattering and/or tangling of concerns arising from the inability of the selected decomposition to modularise them effectively.	
28.		Crosscutting is a structural relationship between representations of concerns. (Crosscutting is a different concept from scattering and tangling.)	
29.		Crosscutting is the occurrence of scattering and tangling of concerns involving a common module.	
30.	Scattering	No explicit definition in Filman et al. [18]	
31.		Scattering is the occurrence of elements that belong to one concern in modules encapsulating other concerns.	
32.		Scattered concern is a concern which cannot be expressed as a single abstraction within the adopted language (Here the term language may refer to req. Specification, analysis, architecture specification, implementation languages, etc.)	
33.		Scattering is the occurrence of the representation of one concern in multiple modules.	
34.	Tangling	Tangling occurs when the code for the implementation of concerns is intermixed. [18]	
35.		Tangling is the occurrence of multiple concerns mixed together in one module.	
36.		Tangled concern is a concern which cannot be expressed as a distinctive abstraction within the adopted language; its definition is not separable from the definition of other concern(s).	
37.		Tangling is the occurrence of the coexistence of representations of multiple concerns in one module.	
38.	Aspect	An aspect is a modular unit designed to implement a concern. [18]	
39.		An aspect is a unit for modularising an otherwise crosscutting concern.	
40.		An aspect is a modularization of a concern.	
41.	Advice	An advice is the behaviour to execute at a join point. [18]	
42.		Advice is an aspect element, which augments or constrains other concerns at join points matched by a pointcut expression.	
43.		An advice is an artefact that augments or constraints concerns at join points.	
44.	Pointcut (Designator)	A Pointcut Designator describes a set of join points. [18]	
45.		A pointcut is a predicate that matches join points. More pre-	

		cisely, a pointcut is a relationship from JoinPoint -> boolean, where the domain of the relationship is all possible join points.	
46.		A pointcut is a selector of join points.	
47.	Join Point	A Join Point is a well-defined place in the structure or execution flow of a program where additional behaviour can be attached. [18]	
48.		A join point is a point of interest in some artefact in the software lifecycle through which two or more concerns may be composed.	
49.	Join Point Model	A Join Point Model (the kind of join points allowed) provides the common frame of reference to enable the definition of the structure of aspects. [18]	
50.		Join point model defines the kinds of join points available and how they are accessed and used.	

6.4. Conclusion

In this chapter, we described an initial version of common AOSD terminology. We started with a selection of core terms and their definitions, primarily based on the book by Filman et al. (2005) [18]. In a workshop and in reviews, we came up with revised definitions and some alternative definitions. It is hard to reach general agreement on this common terminology because the definitions are usually not consistent. This initial version of the glossary should be discussed and improved during the AOSD-Europe project. It should be augmented with other terms to make it comprehensive. It should be made consistent with the common taxonomy - the conceptual model - for AOSD. The current version of the common AOSD glossary is summarized Chapter 1 and in Appendix B.

7. Proposal for a Conceptual Framework for Crosscutting

In this chapter, we describe a proposal for a conceptual framework for crosscutting. The framework focuses on crosscutting and related concepts tangling and scattering. Eventually, the terminology in the glossary (previous chapter) and the conceptual framework (this chapter) should be consistent which each other. This consistency should evolve during the project. (A detailed version of this chapter - including a formalization of the definitions - is available from the authors).

7.1. Introduction

In chapter 3, we described the relation between the glossary (terminology) and taxonomy (conceptual domain model) in an ontology. In this chapter we describe a proposal for a (part of a) taxonomy, a conceptual framework with concepts and definitions related to crosscutting. The framework is developed based on analysis and discussion of definitions of common AOSD terms in the previous chapter, and on some other publications (Masuhara & Kiczales (2003) [30] and of Mezini & Ostermann (2003) [35])

In chapter 3, we described the procedure in the Collaborative Approach to ontology development. For the conceptual framework, we followed the following procedure:

- *Preparation*: The design criterion is that the framework should contain consistent and accurate definitions. These are the requirements explained in section 3.2.1.
- *Anchoring*: The initial set of definitions was based the glossary of common AOSD terms from the previous chapter. Moreover, the ideas came from the AOSD Workshop in Darmstadt, in the session of the Analysis and Design Lab in a presentation by Tekinerdogan.
- *Iterative improvement*: The initial set of terms was reviewed by participants of the Workshop on AOSD Terminology (University of Twente, Enschede), and later by other participants from the AOSD-Project. There is no agreement on this framework.
- *Application*: The usability of the definitions in the conceptual framework should be verified in the other labs of the AOSD-Europe project (e.g. traceability analysis based on cascading of crosscutting patterns).

Overview of chapter

We describe some general issues about the use of the concept crosscutting, and we introduce the crosscutting pattern with some considerations about crosscutting, tangling and scattering (section 7.2).

7.2. Crosscutting

One of the key principles in AOSD is Separation of Concerns (SOC). This principle is described in many publications. Related with this principle is the problem of crosscutting concerns. The concept crosscutting is usually used in an informal way, sometimes leading to ambiguous statements, which may lead to confusion:

" .. the term "crosscutting concerns" is often misused in two ways: To talk about a single concern, and to talk about concerns rather than representations of concerns. Consider "synchronization is a crosscutting concern": we don't know that synchronization is crosscutting unless we know what it crosscuts. And there may be representations of the concerns involved that are not crosscutting. The reason we still say "crosscutting concern" is that when we do so, we are relying on the surrounding dominant decomposition. " (Kiczales, 2005 [26])

Crosscutting is usually described in terms of scattering and tangling (see the previous Chapter 6). However, the distinction between these concepts is vague. We propose a more formal description of crosscutting. This description of crosscutting is similar to some descriptions in the work of Masuhara & Kiczales (2003) [30] and of Mezini & Ostermann (2003) [35].

We propose a Crosscutting Pattern, in which 'elements' in one level (we call this level source) are related to 'elements' in another level (called the target). We use the neutral term 'element' here to denote modules, modular artefact, or artefacts from the usual definitions. We use the term *pattern* as in design patterns (Gamma et al, 1995 [21]), in the sense of being a general description of frequently encountered situations (e.g. Masuhara et al. [30] , Mezini et al.[35], Filman et al. [18]): in these descriptions we have phrases as "one thing *with respect to* another thing".

In our proposed conceptual framework, the **proposition** is that tangling, scattering and crosscutting can only be defined in terms of 'one thing' *with respect to* 'another thing' (in our framework: source *with respect to* target)
In our conceptual framework, we argue that tangling, scattering and crosscutting can be defined as clearly separate cases (of mappings between source and target). We define crosscutting as a specific combination of tangling and scattering.
In our proposed conceptual framework, the **proposition** is that tangling and scattering are necessary but not sufficient conditions for crosscutting.
The rationale for disentangling these three concepts is that there may be different solutions for each of these situations.

Note. A similar view on crosscutting can be found in Masuhara & Kiczales. (2003) [30]. In our terminology, there is a source consisting of A and B and a target X (see Figure 21). Crosscutting is as an intersection of projections.

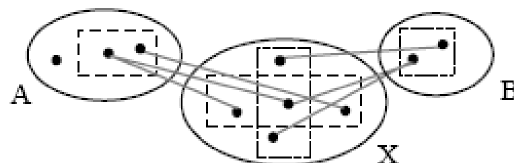


Figure 21. Crosscutting of modules

Crosscutting is defined as follows [30]:

> For a pair of modules m_A and m_B we say that *m_A crosscuts m_B with respect to X* if and only if their projections onto X intersect, and neither of the projections is a subset of the other.

The *intersection* in this definition is similar to our notion of crosscutting.

In some cases, it is possible to avoid tangling, scattering and crosscutting by choosing another decomposition of source and target. The possibilities are determined by the expressive power of the languages in which the source and target are expressed.

"Crosscutting models are themselves not the problem, ... The problem is that our languages and decomposition techniques do not (properly) support crosscutting modularity." (Mezini & Ostermann, 2003 [35])

We explain the role languages in an extension to the crosscutting pattern. In case where limitations in the expressive power of the languages are the cause of tangling, scattering and crosscutting we propose to use the terms *intrinsic tangling*, *intrinsic scattering* and *intrinsic crosscutting*.

In our conceptual model, we make a clear distinction between on hand side inter-level relationships, and on the other hand, the intra-level relationships between elements (related to the concept *coupling*).

We formalized the conceptual model by giving extensional definitions of the concepts. We used matrices to visualize the definitions.

7.3. Conclusion

In this chapter, we described a proposal for a conceptual framework for crosscutting as part of a general taxonomy for AOSD. We introduced a crosscutting pattern with a mapping from a source to a target. With source and target, we abstracted from specific levels or phases in software development. In the framework, we defined crosscutting, tangling and scattering as separated cases based on different mappings between source and target. However, these definitions are not common in the AOSD community and are subject to further research. The proposed definitions are similar to definitions of crosscutting in some other publications, e.g. Masuhara & Kiczales (2003) [30]. The most important criterion for the conceptual framework for crosscutting is to define consistent and precise terminology. Therefore, we introduced new definitions in our framework. However, it is hard to agree on a new conceptual framework because the concepts are usually not commonly accepted. The taxonomy is not complete: the conceptual framework summarized here focuses on crosscutting and related concepts. Eventually, the terminology in the common AOSD glossary and the conceptual framework should be consistent which each other. This consistency should evolve during the project.

8. Conclusion

In this chapter, we summarize the activities of the Common Foundation task. We give an evaluation of the process and of the resulting artefacts: the glossaries and taxonomies. We conclude with some recommendations.

Summary

The goal of the Common Foundation task in the AOSD-Europe project is to provide an ontology of aspect-oriented concepts. The common foundation is a critical success factor for the project: *Sharing a common terminology and associated conceptual model is a key contributing factor to effective technical discussions*.

In this report, we described the development of a Common Foundation for AOSD.

In Chapter 1, we gave a summary of the definitions of an initial set of AOSD common terms.

In Chapter 2, we described the goals of the Common Foundation, the milestones in the task, and the different perspectives from which the report can be read.

In Chapter 3, we described a disciplined process to the development of the common foundation. This ontology engineering process is similar to the development process of software, with requirements, a design and implementation phase of the ontology. We described the requirements for the Common Foundation, among others: comprehensive, general acceptable, consistent, unambiguous and accurate. For the design, we selected the Common Warehouse Metamodel, with a distinction between a glossary with the definition of terms (terminology), and a taxonomy with concepts and relations between concepts (conceptual domain model). We selected a collaborative approach to ontology development. We described web-based support for the development of the glossaries. Furthermore, we introduced the terminology for ontology's that is used in the other parts of this report.

In Chapter 4, we described terminology and a taxonomy for AspectJ. We selected AspectJ as a major representative of aspect languages. In a Pilot study, we used AspectJ to set up the infrastructure for ontology development and to examine representations for taxonomies. The goal of the terminology and taxonomy of AspectJ is to use this as a base for the generalization of terms and concepts within the Common Foundation. We started with a large initial glossary with 70 terms based on the book by Ladded, 2003 [28]. It appeared difficult to cope with so many definitions at the same time. It was decided to split the glossary in three categories: basic terms, advanced terms and general AOSD terms. Changes in definitions were tracked in the Modification Request System. Most changes were recorded on the definitions of Aspect, Advice, Concern, Crosscutting Concern, and Join Point. The current AspectJ glossary contains more than 100 terms and definitions. For the taxonomy, we used views to show how important concepts in AspectJ are related with other concepts. The

definitions in the glossary and conceptual model in the taxonomy should evolve further during the AOSD-Europe project.

In Chapter 5, we described a glossary and taxonomy for ComposeStar. We selected ComposeStar to represent another approach in aspect languages than AspectJ, namely the composition filters approach. The goal of the terminology and taxonomy of ComposeStar is to use this, besides the terminology and taxonomy for AspectJ, as a base for the generalization of terms and concepts within the Common Foundation. The current ComposeStar glossary contains only 11 basic terms and 4 general AOSD terms. During the discussions in the ComposeStar project, it appeared difficult to capture the state-of-the-art of terminology on composition filters, and not to mix it with new research developments, and not to mix it with commonly used AspectJ terminology. It appeared to be a very useful activity for the ComposeStar project to set up such a glossary. The definitions and conceptual models will be developed further as part of the ComposeStar project.

In Chapter 6, we described an initial version of common AOSD terminology. We started with a selection of core terms and their definitions, primarily based on the book by Filman et al. (2005) [18]. In a workshop and in reviews, we came up with revised definitions and some alternative definitions. It is hard to reach general agreement on this common terminology because the definitions are usually not consistent. This initial version of the glossary should be discussed and improved during the AOSD-Europe project. It should be augmented with other terms to make it comprehensive. It should be made consistent with the common taxonomy - the conceptual model - for AOSD.

In Chapter 7, we described a proposal for a conceptual framework for crosscutting as part of a general taxonomy for AOSD. We introduced a crosscutting pattern with a mapping from a source to a target. With source and target, we abstracted from specific levels or phases in software development. In the framework, we defined crosscutting, tangling and scattering as separated cases based on different mappings between source and target. However, these definitions are not common in the AOSD community and are subject to further research. The proposed definitions are similar to definitions of crosscutting in some other publications, e.g. Masuhara & Kiczales (2003) [30]. The most important criterion for the conceptual framework for crosscutting is to define consistent and precise terminology. Therefore, we introduced new definitions in our framework. However, it is hard to agree on a new conceptual framework because the concepts are usually not commonly accepted. The taxonomy is not complete: the conceptual framework summarized here focuses on crosscutting and related concepts. Eventually, the terminology in the common AOSD glossary and the conceptual framework should be consistent which each other. This consistency should evolve during the project.

Evaluation of the process

We used a disciplined approach to ontology development, with explicit requirements, ontology design and implementation. The last phase of ontology development, verification and deployment, should be pursued further in the AOSD-Europe project. We selected a col-

laborative approach to ontology development. It is sometimes hard to involve other partners in this collaborative process due to time constraints and deadlines in other work packages. However, this involvement is essential for reaching consensus on the ontology. We introduced a web-based infrastructure for the support of the glossary development. It was used in several phases and allowed tracking of changes and rationale of the modifications. However, this support is less easy to use than just email. Communication by email was used in more turbulent phases of the discussion of terminology in this common foundation task.

Evaluation of the products

We described glossaries and taxonomies for AspectJ and ComposeStar. We used these glossaries for generalization to the common AOSD glossary and taxonomy. The glossaries and taxonomies are first versions and should evolve during the project. They do not completely cover the whole domain of AOSD. Moreover, the consistency between glossaries and taxonomies should be improved. This requires further involvement of other partners in the AOSD-Europe project.

Dilemma

During the development of the AOSD ontology, we encountered conflicting requirements for the ontology: to be general acceptable (consensus), to be conflict-free (consistency), to be unambiguous and accurate. That situation is captured in the following dilemma:

It is hard to reach agreement on common terminology because this terminology is usually not consistent. It is hard to reach agreement on consistent terminology because this terminology is usually not common.

In this report, we presented the first public version of the common glossary for AOSD based on a disciplined approach to ontology development. There should be regular new updates of this common glossary based on reviews and further discussion. Furthermore, we proposed a part of the common taxonomy for AOSD, the conceptual framework for crosscutting. This framework should be discussed and extended to a complete taxonomy for AOSD. The consistency between the common glossary and the common taxonomy should evolve during the AOSD-Europe project. We conclude this report with the following recommendations.

Recommendations

1. The AOSD-glossary should be made widely available, e.g. on the AOSD.NET website `http://aosd.net/wiki/index.php?title=Glossary`
2. Deliverables of AOSD-Europe should include an explicit glossary with definitions of important AOSD terms used in the deliverable.
3. Definitions in the deliverables should be evaluated regularly in order to achieve continuous improvement of the Common Foundation for AOSD.

Acknowledgement

The authors - Klaas van den Berg (University of Twente), Jose Maria Conejero (University of Extremadura, visiting researcher at the University of Twente) and Ruzanna Chitchyan (University of Lancaster) - would like to acknowledge the contribution of a large number of people to this Common Foundation (in alphabetical order):
Adrian Coyler, Awais Rashid, Bedir Tekinerdogan, Christa Schwanninger, Frans Sanen, Gurcan Gulesir, Istvan Nagy, Lodewijk Bergmans, Maja d'Hondt, Mario Sudholt, Mehmet Aksit, Mira Mezini, Neil Loughran, Pascal Durr, Shmuel Katz, Siobhan Clarke, and others.

References

[1] ANSI/IEEE 830 (1984). Guide to Software Requirements Specification.

[2] AOSD (n.d.), Conferences on Aspect Oriented Software Development. See http://aosd.net/archive/index.php

[3] AOSD-Europe (2004). IST Project Proposal 004349, Annex I - Description of Work, 1 September 2004.

[4] Apache Tomcat 4.1. See http://jakarta.apache.org/tomcat/

[5] Borland Together 6.1 (n.d.) See http://www.borland.com/products/

[6] Borland Together Designer 2005. See http://www.borland.com/products/downloads/download_together.html

[7] Bugzero (n.d.). See http://www.websina.com/bugzero/

[8] Bugzilla (n.d.). See http://www.bugzilla.org/

[9] Chavez, C., Lucena, C.J.P. (2003). A Theory of Aspects for Aspect-Oriented Software Development. SBES 03 17º. Simpósio Brasileiro de Engenharia de Software, Manaus, Editora da Universidade Federal do Amazonas, 2003. p.19 - 34

[10] ComposeStar See http://janus.cs.utwente.nl:8000/twiki/bin/view/Composer/WebHome

[11] Concern Manipulation Environment (CME). See http://www.research.ibm.com/cme/

[12] Copi, I.M. & Cohen, C. (1998), Introduction to Logic, 10th edition, Prentice Hall

[13] CWM (2003). Common Warehouse Metamodel (CWM) Specification, March 2003, Version 1.1, Volume 1, formal/03-03-02

[14] Czarnecki, K., Eisenecker, U.W. (2000), Generative Programming, Addison-Wesley, Reading

[15] Early Aspects (n.d.). See http://www.early-aspects.net/

[16] Fernandez, M., A. Gomez-Perez, and N. Juristo. METHONTOLOGY: From Ontological Art Towards Ontological Engineering. In Workshop on Ontological Engineering - Symposium on Ontological Engineering of AAAI. 1997. Standford, California.

[17] Filman, R. (2004). A Bibliography of Aspect-Oriented Programming. Version 1.1. See http://www.riacs.edu/navroot/Research/TR_pdf/TR_03.01.pdf

[18] Filman, R., et al., *Aspect-Oriented Software Development*. 2004: Addison-Wesley.

[19] Firesmith, D.G. & E.M. Eykholt (1995). Dictionary of Object Technology. The Definitive Desk Reference. Sigs Books, New York

[20] FAOL (n.d.). Workshops on Foundations of Aspect-Oriented Languages. See http://www.cs.iastate.edu/~leavens/FOAL/index-2005.shtml

[21] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design patterns. Elements of reusable object-oriented software: Addison-Wesley.

[22] Gardner, T., et al. A review of OMG MOF 2.0 Query/Views/Transformations submissions and recommendations towards the final standard. In 1st International Workshop on Metamodeling for MDA. 2003. York, UK.

[23] Gruber, T. (n.d.). What is an ontology. See http://www-ksl.stanford.edu/kst/what-is-an-ontology.html

[24] Holsapple, C.W. & Joshi, K.D. (2002). A Collaborative Approach to Ontology Design. Communication of the ACM, February 2002, Vol 45, No 2, p. 42 - 47.

[25] ISO/IEC9126 (1991). *International Standard ISO/IEC 9126. Information technology -- Software product evaluation -- Quality characteristics and guidelines for their use*, International Organization for Standardization, International Electrotechnical Commission, Geneva (for a summary see http://www.cse.dcu.ie/essiscope/sm2/9126ref.html)

[26] Kiczales, G. *Crosscutting*. AOSD.NET Glossary 2005 [cited; Available from: http://aosd.net/wiki/index.php?title=Crosscutting

[27] Kurtev, I. and K.v.d. Berg. A Synthesis-Based Approach to Transformations in an MDA Software Development Process. In Model Driven Architecture: Foundations and Applications. 2003. Enschede

[28] Laddad, R. (2003). AspectJ in Action, Practical Aspect-Oriented Programming. Manning, Greenwich

[29] Lopes, C.V. and S.K. Bajracharya. An analysis of modularity in aspect oriented design. In 4th international conference on Aspect-Oriented Software Development. 2005. Chicago, Illinois

[30] Masuhara, H. and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In ECOOP 2003. 2003. Darmstadt

[31] MDA (2003). MDA Guide Version 1.0.1, document number omg/2003-06-01.

[32] Mehner, K . & Rashid, A. (2003). Towards a Generic Model for AOP (GEMA)., University of Lancaster, Technical Report No. CSEG/1/03.
see http://www.comp.lancs.ac.uk/computing/aose/papers/FASOP_TechRep.pdf

[33] Mehner, K. and Rashid, A. (2003) *GEMA: A Generic Model for AOP*. Belgian and Dutch Workshop on AOP, Twente, The Netherlands.

[34] Mellor, S.J. and M.J. Balcer, Executable UML, A Foundation for Model-Driven Architecture. 2002: Addison-Wesley.

[35] Mezini, M. and K. Ostermann. Modules for Crosscutting Models. In 8th International Conference on Reliable Software Technologies. 2003. Toulouse, France: LNCS 2655

[36] Microsoft Office Access 2003. See http://www.microsoft.com/office/access/

[37] Noy, N.F. and McGuinness, D.L. (n.d.). Ontology Development: A Guide to Creating Your First Ontology. See http://www.ksl.stanford.edu/people/dlm/papers/ontology101/ontology101-noy-mcguinness.html

[38] OWL (2004). Web Ontology Language Overview W3C Recommendation, 10 February 2004. See http://www.w3.org/TR/2004/REC-owl-features-20040210/

[39] PMBOK (2000). Project Management Body of Knowledge Guide 2000. See http://www.pmi.org/info/PP_PMBOKGuide2000Excerpts.pdf

[40] Rashid, A., A. Moreira, and J. Araujo. Modularisation and Composition of Aspectual Requirements. In Second AOSD Conference. 2003. Boston

[41] Rumbaugh, J., Jacobson, I. & Booch, G. (1999). The Unified Modelling Language Reference Manual. Addison-Wesley

[42] Shanks, G., Tansley, E. & Weber, R. (2003). Using Ontology to Validate Conceptual Models. Communication of the ACM, October 2003, Vol 46, No 10, p. 85 - 98.

[43] Skulmoski, G. (2002). Shifting Gears: the De Facto Global Standard for Project Management.
See http://www.pmi-lakeshore.org/present_20020311_Shifting_Gears.ppt

[44] Sutton, S. and I. Rouvellou, Concern Modeling for Aspect-Oriented Software Development, in Aspect-Oriented Software Development, R. Filman, et al., Editors. 2004, Addison Wesley

[45] Sutton, S. and I. Rouvellou. *Modeling of Software Concerns in Cosmos*. in *Aspect-Oriented Software Development*. 2002. Enschede, The Netherlands

[46] Tekinerdogan, B. ASAAM: Aspectual Software Architecture Analysis Method. in WICSA 4th Working IEEE/IFIP Conference on Software Architecture. 2004: IEEE

[47] UML (2003). UML 2.0 Superstructure Specification OMG ptc/03-08-02

Appendix A. Glossary with Definition of Ontology Terms

In this appendix we give an overview of the definitions of terms (in alphabetical order) related with ontology development as presented in Chapter 3.

Terms and Definitions

Term	Definition
Collaborative Ontology Development	Collaborative Ontology Development
Common Warehouse Metamodel	The Common Warehouse Metamodel is a model that shows the relation between a taxonomy (domain model) and a glossary (terminology) for a domain.
Concept	A Concept is an item in a taxonomy describing a matter of interest in a certain domain.
Conceptual Framework	A Conceptual Framework is a coherent collection of related concepts
Definition	A Definition is a description of a term or a concept. The definition can be denotative or operational.
Denotative Definition	A Denotative Definition is a definition of a concept (or a term) with the structure: <Concept> is <more general concept> with <specific properties>. Synonym: Genus-difference definition
Diagram	A Diagram is a graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements).
Domain	A Domain is an area of knowledge.
Extensional Definition	An extensional definition of a concept is a definition that describes a collection of 'things' that follow under that definition.
Glossary	A Glossary is a collection of terms in a certain domain and consists terms with their definitions and their relations. Synonym: Terminology
Intensional Definition	An intensional definition of a concept is a definition that states the essential properties of the concept.
Model	A Model is a semantically complete abstraction of a system - from a particular viewpoint - consisting of model elements. A Model element is an abstraction drawn from the system being modelled.
Ontology	An Ontology is a definition of common concepts and relationships used to describe and represent an area of knowledge.
Operational Definition	An Operational Definition of a term (or concept) is a definition that states that the term is correctly applied to a given case if and only if the performance of specified operations

	in that case yields a specified result.
Taxonomy	A Taxonomy is a conceptual model of a certain domain and consists of a collection of concepts with their relations. Synonym: Domain model
Term	A Term is an item in a glossary representing a matter of interest in a certain domain.
View	A View is a model which is completely derived from another model (the base model). A view cannot be modified separately from the model from which it is derived. Changes to the base model cause corresponding changes to the view.

Appendix B. Glossary with Common AOSD Terminology

In this appendix, we give an overview of preferred definitions of common AOSD terms (in alphabetical order) as presented in Chapter 1.

Term	**Definition**
Advice	An advice is an aspect element, which augments or constrains other concerns at join points matched by a pointcut expression.
Aspect	An aspect is a unit for modularising an otherwise crosscutting concern.
Composition	Composition is the integration of multiple modular artefacts into a coherent whole.
Concern	A concern is an interest, which pertains to the system's development, its operation or any other matters that are critical or otherwise important to one or more stakeholders.
Crosscutting	Crosscutting is the scattering and/or tangling of concerns arising from the inability of the selected decomposition to modularise them effectively.
Crosscutting Concern	A crosscutting concern is a concern, which cannot be modularly represented within the selected decomposition. Consequently, the elements of crosscutting concerns are scattered and tangled within elements of other concerns.
Decomposition	Decomposition is the breaking down of a larger problem into a set of smaller problems which may be tackled individually.
Join Point	A join point is a point of interest in some artefact in the software lifecycle through which two or more concerns may be composed.
Join Point Model	A join point model defines the kinds of join points available and how they are accessed and used.
Pointcut	A pointcut is a predicate that matches join points. More precisely, a pointcut is a relationship from JoinPoint -> boolean, where the domain of the relationship is all possible join points.
Scattering	Scattering is the occurrence of elements that belong to one concern in modules encapsulating other concerns.
Separation of Concerns	Separation of Concerns is an in depth study and realisation of concerns in isolation for the sake of their own consistency (adapted from “On the Role of Scientific Thought” by Dijkstra, EWD 447).
Tangling	Tangling is the occurrence of multiple concerns mixed together in one module.
Tyranny of Dominant Decomposition	The Tyranny of the Dominant Decomposition refers to restrictions (or tyranny) imposed by the selected decomposition technique (i.e. the dominant decomposition) on software engineer's ability to modularly represent particular concerns.
Weaving	Weaving: Historically this term is used to refer to the composition of aspects with other concerns in the system. See composition.

Appendix C. AspectJ - Glossary

In this appendix, we give an overview of the definitions of AspectJ terms (in alphabetical order) as presented in Chapter 4.

The terms are given in four categories:
- Advanced terms
- Basic terms
- General AOSD terms
- References

The terms are shown in a UML Package diagram (see Figure 22).

State: February 22, 2005

Figure 22. AspectJ concepts and views (UML Package Diagram)

	Advanced Terms and Definitions
Abstract Aspect	An Abstract Aspect is an aspect that contains methods and pointcuts, which may be abstract methods and abstract pointcuts. Note: An abstract aspect by itself does not cause any weaving to occur; concrete subaspects must be provided to do so.
Abstract Pointcut	An Abstract Pointcut is a pointcut defined within an abstract aspect. This pointcut is defined without specify any join point. The join points connected to the pointcut will be defined in a concrete pointcut.
Advice Execution Join Point	An Advice Execution Join Point is a category of join points in which the scope is the execution of an advice.
Advice Execution Pointcut	An Advice Execution Pointcut is a kinded pointcut based on matching of join points in the Advice execution join point category.
After Returning Advice	An After Returning Advice is an after advice that is only executed when the execution of the join point which it affects finishes succesfully.
After Throwing Advice	An After Throwing Advice is an after advice that is only executed when the execution of the join point which it affects finishes throwing an exception.
Argument Pointcut	An Argument Pointcut is a conditional pointcut in which join points are captured based on the argument type of join point.
Binary Pointcut Operator	A Binary Operator requires two operands. AspectJ supports AND (&&) and OR(||) operators. Combining two pointcuts with the || operator causes the selection of join points that match either of the pointcuts, whereas combining them with the && operator causes the selection of join points matching both the pointcuts.
Class Initialization Join Point	A Class Initialization Join Point is a category of join points in which the scope is the loading of a class, including the static portion.
Class Initialization Pointcut	A Class Initialization Pointcut is a kinded pointcut based on matching of join points in the Class initialization join point category.
Compile-time Declaration	A Compile-time Declaration is a static crosscutting instruction that adds compile-time warnings and errors upon detecting certain usage patterns.
Compile-time Error and Warning Declaration	Compile-time Error and Warning Declaration is a form of static crosscutting in which these errors and warnings are declared based on certain usage patterns.
Composite Pointcut Signature Pattern	A Composite Pointcut Signature Pattern is the definition of a pointcut that use the wmatching pattern mechaism in order to capture join points that share common characteristics in their signatures. A Composite Pointcut Signature Pattern is also know as Property-Based Pointcuts.
Concrete Aspect	A Concrete Aspect is an aspect that contains only concrete methods and concrete pointcuts (and no abstract methods and no abstract pointcuts).
Concrete Pointcut	A Concrete Pointcut is a pointcut that implements an abstract pointcut. The concrete pointcut defines the join point connected to the abstract one. A Concrete Pointcut can be a primitive or a composite pointcut.

Conditional Pointcut	A Conditional Pointcut is a pointcut in which matching of join points is according to the matching of a prescribed condition. There are 4 types of conditional pointcuts: Control-flow based pointcut, Lexical-structure based pointcut, Execution object pointcut, Argument pointcut
Constructor Call Join Point	A Constructor Call Join Point is a category of join points in which the scope is the invocation or call of the constructor.
Constructor Call Pointcut	A Constructor Call Pointcut is a kinded pointcut based on matching of join points in the call execution join point category.
Constructor Execution Join Point	A Constructor Execution Join Point is a category of join points in which the scope is the body of the constructor.
Constructor Execution Pointcut	A Constructor Execution Pointcut is a kinded pointcut based on matching of join points in the Constructor execution join point category.
Constructor Join Point	A Constructor Join Point is a category of join points in which the scope is the constructor of a class. There are Constructor execution join points and Constructor call join points.
Constructor Pointcut	A Constructor Pointcut is a kinded pointcut based on matching of join points in the Constructor join point category. There are 2 types of Constructor pointcuts: Constructor execution pointcut, Constructor call pointcut.
Constructor Signature Pattern	The Constructor Signature Pattern allows the pointcuts to identify call and execution join points in constructors that match the simple pointcut signature patterns.
Control-flow Based Pointcut	A Control-flow Based Pointcut is a conditional pointcut in which join points are captured based on the control flow of join points captured by another pointcut.
Dynamic Crosscutting Rule	A Dynamic crosscutting rule is formed by pointcuts and advice, where pointcuts specify the weaving rules (i.e. where to weave in additional logic) and advice specify the required additional logic.
Dynamic Pointcut	A Dynamic Pointcut is a pointcut that exposes context, because it can operate with run time information. It is also know as Dynamically Determinable Pointcut. The following list enumerates the Dynamic Pointcuts: cflow, cflowbelow, if, this, target and args.
Exception Handler Execution Join Point	An Exception Handler Execution Join Point is a category of join points in which the scope is the handler block of an exception type.
Exception Handler Execution Pointcut	An Exception Handler Execution Pointcut is a kinded pointcut based on matching of join points in the Exception handler execution join point category
Exceptioning Softening	Exceptioning Softening is a form of static crosscutting in which checked exceptions thrown by specified pointcuts are treated as unchecked ones.
Execution Object Pointcut	An Execution Object Pointcut is a conditional pointcut in which join points are captured based on the types of objects at execution time.
Exposed Join Point	An Exposed Join Point is a join point that can be selected in pointcuts.
Field Access Join Point	A Field Access Join Point is a category of join points in which the scope is the access to an instance or class member of a class. There are Field read access join points and Field write access join points
Field Access	A Field Access Pointcut is a kinded pointcut based on matching of join points in the

Pointcut	Field access join point category. There are 2 types of Field access pointcuts: Field read access pointcut, Field write access pointcut.
Field Read Access Join Point	A Field Read Access Join Point is a category of join points in which the scope is the read access to an instance or class member of a class.
Field Read Access Pointcut	A Field Read Access Pointcut is a kinded pointcut based on matching of join points in the Field read access join point category.
Field Signa-ture Pattern	The Field Signature Pattern allows to capture join points corresponding to read or write access to specified field.
Field Write Access Join Point	A Field Write Access Join Point is a category of join points in which the scope is the write access to an instance or class member of a class.
Initialization Join Point	An Initialization Join Point is a kind of join point associated with the initializtion of clases and objects.
Initialization Pointcut	Initialization Pointcut is the group of pointcuts that can match join points regarding to the initialization of classes and objetcts.
Invocation Join Point	An Invocation Join Point is a kind of join point associated with the call or execution of methods or other elements.
Invocation Pointcut	Invocation Pointcut is the group of pointcuts that can match join points when meth-ods or other elements are called or executed.
Join Point Category	A join point category is a collection of join points with a specific scope.
Kinded Pointcut	A Kinded Pointcut is a pointcut in which matching of join points is according to the category to which a join point belongs. There are 8 types of kinded pointcuts: Method pointcut, Constructor pointcut, Field access pointcut, Exception handler execution pointcut, Class initialization pointcut, Object initialization pointcut, Object pre-initialization pointcut, Advice execution pointcut.
Lexical-structure Based Point-cut	A Lexical-structure Based Pointcut is a conditional pointcut in which join points are captured inside a lexical scope of specified classes, aspects and methods.
Matching Pattern	The Matching Pattern is the mechanism to group together join points specified by multiple signatures. This mechanism used language artefacts like wildcards *, .. and +.
Member Introduction	Member Iintroduction is a form of static crosscutting in which members (data fields and methods) are added to specified classes and interfaces.
Method Call Join Point	A Method Call Join Point is a category of join points in which the scope is the call of the method.
Method Call Pointcut	A Method Call Pointcut is a kinded pointcut based on matching of join points in the Method call join point category.
Method Exe-cution Join Point	A Method Execution Join Point is a category of join points in which the scope is the execution of the method body.
Method Exe-cution Point-cut	A Method Execution Pointcut is a kinded pointcut based on matching of join points in the Method execution join point category.
Method Join Point	A Method Join Point is a category of join points in which the scope is the method of a class. There are Method execution join points and Method call join points.
Method Pointcut	A Method Pointcut is a kinded pointcut based on matching of join points in the Methods join point category. There are 2 types of Method pointcuts: Method exe-cution pointcut, Method call pointcut.
Method Sig-	The Method Signature Pattern allows the pointcuts to identify call and execution

nature Pattern	join points in methods that match the simple pointcut signature patterns.
Normal After Advice	A Normal After Advice is an after advice that will be executed after the join point it affects, regardless of the outcome.
Object Initialization Join Point	An Object Initialization Join Point is a category of join points in which the scope is the initialization of an object starting from the return of a parent class' constructor until the end of the first called constructor.
Object Initialization Pointcut	An Object Initialization Pointcut is a kinded pointcut based on matching of join points in the Object initialization join point category.
Object Pre-initialization Join Point	An Object Pre-initialization Join Point is a category of join points in which the scope is the passage from the constructor that was called first to the beginning of its parent constructor.
Object Pre-initialization Pointcut	An Object Pre-initialization Pointcut is a kinded pointcut based on matching of join points in the Object pre-initialization join point category.
Per-control-flow Aspect Association	A Per-control-flow Aspect Association is an aspect association in which one instance of the aspect is created for each control-flow matching the association specification.
Per-object Aspect Association	A Per-object Aspect Association is an aspect association in which one instance of the aspect is created for each object with per-object state.
Per-virtual-machine Aspect Association	A Per-virtual-machine Aspect Association is an aspect association in which one instance of the aspect is created with shared state. This association is also used by default when no association is defined.
Pointcut Designator	A Pointcut Designator is an identifier of a pointcut either by name or by an expression.
Simple Pointcut Signature Pattern	A Simple Pointcut Signature Pattern is a pointcut signature pattern which capture a single join point. A composite pointcut signature pattern is realized by the combination of some Simple Pointcut Signature Pattern elements. There are 4 types of Simple Pointcut Signature Patterns: Type signature pattern, Method signature pattern, Constructor signature pattern, Field signature pattern.
Static Pointcut	A Static Pointcut is a pointcut which operates only on compile time information. It is also know as statically determinable pointcut. The primitive pointcut that fit the description are: call(), execution(), advice execution(), get(), set(), handler(), intialization(), static initialization(), within() and within code().
Type Signature Pattern	A Type Signature Pattern in a pointcut specifies the join points in a type, or a set of types, at which you want to perform some crosscutting action.
Unary Pointcut Operator	A Unary Operator requires only one operand. AspectJ supports only one unary pointcut operator: the negation (!). The negation operator allows the matching of all join points except those specified by the pointcut.
	Basic Terms and Definitions
Advice	An Advice is a method-like construct that provides a way to express crosscutting action at the join points that are captured by a pointcut. There are three kinds of Advice: Before advice, After advice, Around advice.
After Advice	An After Advice is an advice that executes after the execution of the captured join point.

Around Ad-vice	An Around Advice is an advice that can bypass the execution of the captured join point, execute it with different argument, execute it multiple times, and/or perform additional execution before and after the join point.
Aspect	An Aspect is a class-like unit in AspectJ for modularising crosscutting concerns. Aspects containe code that expresses the weaving rules for both dynamic and static crosscutting. Unlike classes, AspectJ aspects are not directly instantiated via 'new' expression, but are automatically created by AspectJ weaver.
Aspect As-sociation	An Aspect Association is a specification of the way the aspect's state is bound. There are 3 categories of aspect associations: Per-virtual-machine aspect associa-tion, Per-object aspect association, Per-control-flow aspect association.
Aspect In-heritance	Aspect Inheritance defines the generalization/specialization relation between an aspect and a subaspect.
Aspect In-stantiation	Unlike class expressions, [AspectJ] aspects are not instantiated with new expres-sions. Rather, aspect instances are automatically created to cut across programs. The multiplicity of aspect creation is defined by aspect association.
Aspect Precedence	Aspect Precedence controls the advice execution order, which is the order in which the advice is applied, in cases that advice in more than one aspect applies to a join point.
Base Aspect	A Base Aspect is an aspect that is specialized in one or more subaspects.
Base Code	The Java classes and interfaces comprising the implementation of the original ob-ject-oriented decomposition to which AspectJ aspects will be applied. Note that the term base code is also used outside the AspectJ context, where it may refer to a non-Java and non-object-oriented program.
Before Ad-vice	A Before Advice is an advice that executes before the execution of the captured join point.
Composite Pointcut	A Composite Pointcut is a concrete pointcut formed by the union of primitive point-cuts. A Composite Pointcut matchs some different sets of join points.
Dynamic Crosscutting Structure	Dynamic Crosscutting Structure is a set of language constructs which allows the aspect to express dynamic crosscutting actions. The Dynamic Crosscutting Structure can be composed of advices and pointcuts.
Inter-type Declaration	An Inter-type Declaration is a declaration by an aspect of members that are (as a result)associated with other types. Examples are: - inter-type method declaration - inter-type constructor declaration - inter-type field declaration
Introduction	An Introduction is a static crosscutting instruction that introduces changes to the static structure of the system (classes, interfaces, aspects)
Join Point	A Join Point is an identifiable point in the execution of a program in a certain con-text. Currently the following join point types are used: Method execution and Method call (termed Method join points together); Constructor execution and Con-structor call (termed Constructor join points together); Field read access and Field write (termed Field access join points together); Exception handler execution; Class initialization; Object initialization; Object pre-initialization; Advice execution.
Pointcut	A Pointcut is a program construct that selects join points by matching certain char-acteristics and collects context at those points. A pointcut can be an abstract or a concrete pointcut.

Pointcut Operator	A Pointcut Operator is an operator used to form more complex pointcuts by combining simple pointcuts. In AspectJ, there is a one unary pointcut operator - the negation (!), and tow binary pointcut operators - OR and AND (||, &&). The precedence between these operators is the same as in plain Java.
Pointcut Signature Pattern	A Pointcut Signature Pattern is a specification of the places where to capture join points based on signatures of classes, interfaces, and methods. A Pointcut Signature Pattern can be a simple pointcut signature pattern or a composite pointcut signature pattern.
Primitive Pointcut	A Primitive Pointcut is a concrete pointcut that selects a set of join points by matching certain characteristics and collects context at those points. There are two types of primitive pointcuts: kinded pointcuts and conditional pointcuts.
Static Crosscutting Structure	Static Crosscutting Structure is a set of language constructs which allows the aspect to express some static crosscutting actions. The Static Crosscutting Structure allows us define the inter-type declaration elements of the aspect.
Subaspect	A Subaspect is an aspect that inherits from a base aspect.
Type-hierarchy Modification	Type-hierarchy Modification is a form of static crosscutting in which the inheritance hierarchy is modified by declaring a superclass and interfaces of an existing class of interface.
	General Terms and Definitions
Aspectual Decomposition	Aspectual Decomposition is the software development phase in which requirements are decomposed into core concerns and crosscutting concerns.
Aspectual Recomposition	Aspectual Recomposition is the software development phase in which recomposition rules are specified by creating aspects and by applying these rules in the weaving process to the implemented concerns.
Compile-time Weaving	Compile-time Weaving is a weaving type in which the weaving occurs at the compilation time.
Composability	Composability is a characteristic of software artefacts (e.g. code, designs) which facilitates integration of multiple modular artefacts into a coherent whole. An alternative definition: composability is the ability to define a new software artefact via construction of two or more artefacts, still preserving the required characteristics of the initial one.
Composition Scheme	A Composition Scheme is the model for composing software from separate entities.
Concern	A Concern is a specific need that must be addressed in order to satisfy the overall system goal. After softeare modularisation apporach has bee selected, there could be 2 types of concerns: Core concern and Crosscutting concern.
Concern Implementation	Concern Implementation is the software development phase in which core concerns and crosscutting concerns are implemented independently.
Context	The Context of a join point contains the information about the current execution of the program (caller object, target object, arguments of methods, etc.)
Core Concern	A Core Concern is a concern that captures the central functionality of a module in a system.

Crosscutting	Crosscutting is the scattering and tangling of concerns arising due to poor support for their modularisation. We distinguish two levels of crosscutting: - Crosscutting at design level is the scattering and tangling of concerns at modeling level. - Crosscutting at implementation level is the scattering and tangling of code which belongs to different concerns.
Crosscutting Concern	A Crosscutting Concern is a concern that captures requirements that cross multiple modules in a system. It should be noted that a concern can be crosscutting in respect with a certain set of modules (e.g. classes) but it can become non-crosscutting if an alternative set of modules is selected (e.g. features). Thus crosscutting exists only in relation to set modularisation constructs.
Dominant Decomposition	Dominant Decomposition is a decomposition of a problem domain which is used by other decompositions of that same problem domain in their definition as a reference (asymmetric). Opposed to nondominant decomposition, which is a decomposition that is equally important as other decompositions (symmetric).
Dynamic Crosscutting	Dynamic Crosscutting is the weaving of new behavior into the execution of a program.
Dynamic Weaving	The Dynamic Weaving is the weaving which can change the aspects used against the base code in run time. This kind of weaving is able to apply aspects to the base code dynamically. The Dynamic Weaving is also know as Run-time weaving.
Field Write Access Pointcut	A Field Write Access Pointcut is a kinded pointcut based on matching of join points in the Field write access join point category.
Join Point Model	A Join point model is a model of a system in which join points are defined and separated in exposed and not-exposed join points in order to prevent implementation-dependent or unstable crosscutting. In addition to exposed join points (e.g. Method Call join points), the join point model defines the way to refer to these join points (e.g. through pointcut), and a mechanism to affect the programme at the selected join points (e.g. through advice).
Link-time Weaving	Link-time Weaving is a weaving type in which the weaving occurs after the compiled primary and aspect byte code is loaded, as the binaries are combined into the runtime state of the Java virtual machine to become ready for execution.
Load-time Weaving	Load-time Weaving is a weaving type in which the weaving occurs when classes are loaded by the classloader. Ultimately, the weaving is at the byte-code level.
Run-time Weaving	Run-time Weaving is a weaving type in which the virtual machine is responsible for detecting join points and loading and execution [of] aspects.
Scattered Code	Scattered Code is code in which the implementation of a single concern is spread over multiple modules.
Scope	The Scope of an aspect instance is the set of join points that have an aspect instance associated with them.
Software System	A Software System is the realization of a set of concerns.
Static Crosscutting	Static Crosscutting is the weaving of modifications into the static structure of the system (classes, interfaces, aspects). There are 4 types of static crosscutting: member introduction, type-hierarchy modification, compile-time error and warning declaration, and exceptioning softening.
Static Weaving	The Static Weaving is the weaving that is not able to change dynamically the aspects used against the base code. There are three kind of Static Weaving: com-

	pile-time weaving, load-time weaving and link-time weaving.
Tangled Code	Tangled Code is code in which the implementation of multiple concerns is handled simultaneously in a single module.
Weaver	A Weaver is a processor that performs the weaving according to weaving rules.
Weaving	Weaving is the process of composing the system from individual core modules by following the weaving rules. There are 2 types of weaving: Static weaving (see Static crosscutting), Dynamic weaving (see Dynamic crosscutting)
Weaving Time	Weaving Time defines the time at which weaving is performed. There are 4 weaving-time categories: Compile-time weaving, Link-time weaving, Load-time weaving, Run-time weaving.
	References
AspectJ Language Semantics, Eclipse	AspectJ Language Semantics, Eclipse
AspectJ Semantics Appendix	AspectJ Project Documentation
Getting Started with AspectJ, Eclipse	Getting Started with AspectJ, Eclipse
Gradecki & Lesiecki, 2003	Gradecki, J.D. & Lesiecki, N. (2003). Mastering AspectJ - Aspect-Oriented Programming in Java. Wiley
Kersten, 2005	Kersten, M. AOP@WORK: AOP tools comparison, Part 1. IBM.
Laddad, 2003	Laddad, R. (2003). AspectJ in Action, Practical Aspect-Oriented Programming. Manning, Greenwich

Appendix D. ComposeStar - Glossary

In this appendix, we give an overview of the definitions of ComposeStar terms (in alphabetical order) as presented in Chapter 5.

The ComposeStar terms are given in four categories:
- Advanced terms
- Basic terms
- General AOSD terms
- References

State: February 22, 2005

	Advanced Terms and Definitions
	None
	Basic Terms and Definitions
Base Object	The Base Object is a regular object excluding filters/filtermodules. It can be implemented in "any" object-based language. When applying Composition Filters this base object will be enhanced by a layer of filters.
Composition Filter Object Model	The Composition Filter Object Model denotes the characteristics of objects according to Composition Filter Approach. In particular, the Composition Filter Object Model extends the base object-oriented model with a layer that allows manipulation of incoming and outgoing messages of objects.
Compositor	Composition Operator.
Concern	A Concern is the principle unit of Compose* that may specify filtermodules, superimposition and the implementation of the base object. (All these elements are optional.) In general, the term 'concern' may refer to classes with or without filters. (That is, if we leave out the filtermodules and superimposition the resulting concern corresponds to a regular class.) When a concern is instantiated we use the term 'concern instance'.
Condition	The Condition represents the state of the system or filtermodule. This state can be dervied from the internals, externals or from another object. The condition is used in the filter expressions to determine whether a filter accepts or rejects.

Condition Binding	This Condition Binding specifies which conditions are superimposed on the objects specified in the selector definition. This is needed to make the conditions explicitly available at the interface of the objects.
External	An External is an object reference contained by a filter module. This object is instantiated using an expression and this instance can be shared by multiple filtermodules. The internals, in combination with the internals, represent the state of the filtermodule.
Filter	A Filter specifies a behavior enhancement - through inspection and manipulation of messages - of an concern instance. A filter is described by a filter specification. There are input filters and output filters. Depending on the state and the current message the filter can have accept or reject behavior. The actions asociated with these behavior depend on the filter type.
Filter	To be removed... (See 1129)
Filter Composition Operator	* an operator that defines how multiple (usually two) filters are connected. * NB: the ordering constraints *also* partially define the composition
Filter Declaration	A Filter Declaration follows a simple, declarative, structured language. It denotes the identifier, type (with optional paramters) of a filter and its filter pattern. It defines how to create a filter instance.
Filter Element Compositor	A Filter Element Compositor is for connecting filterelements within a filter pattern. It influences the evaluation of a filterelement.
Filter Evaluation	Filter Evaluation is the execution of the filter behavior. The filter behaviour is defined by the filter pattern and filter type. The result of the evaluation depends on the given message and, implicitly, the state of system. The evaluation can lead to one or more of the following actions: * change the state of system * change the message * yield 'terminate' and 'continue'
Filter Identifier	The Filter Identifier denotes the name of the filter. The filter can be identified through this name. This name may not be duplicated in the filtermodule, the same name may be present in other filtermodules. A filter can be uniquely identified via a ConcernName.FilterModuleName.FilterName expression.
Filter Module	* the unit of reuse and instantiation of filter behavior. * group of filter definitions that belong conceptually together (e.g. since they perform a single logical task) and have no meaningful function independently * a subcomponent of a concern * In addition to the specification of filters, filter modules may provide some execution context for the filters by specifying internals, externals, methods & conditions that can be used in the filters. * comparable to a group of advices in AspectJ (but without the connected pointcuts)

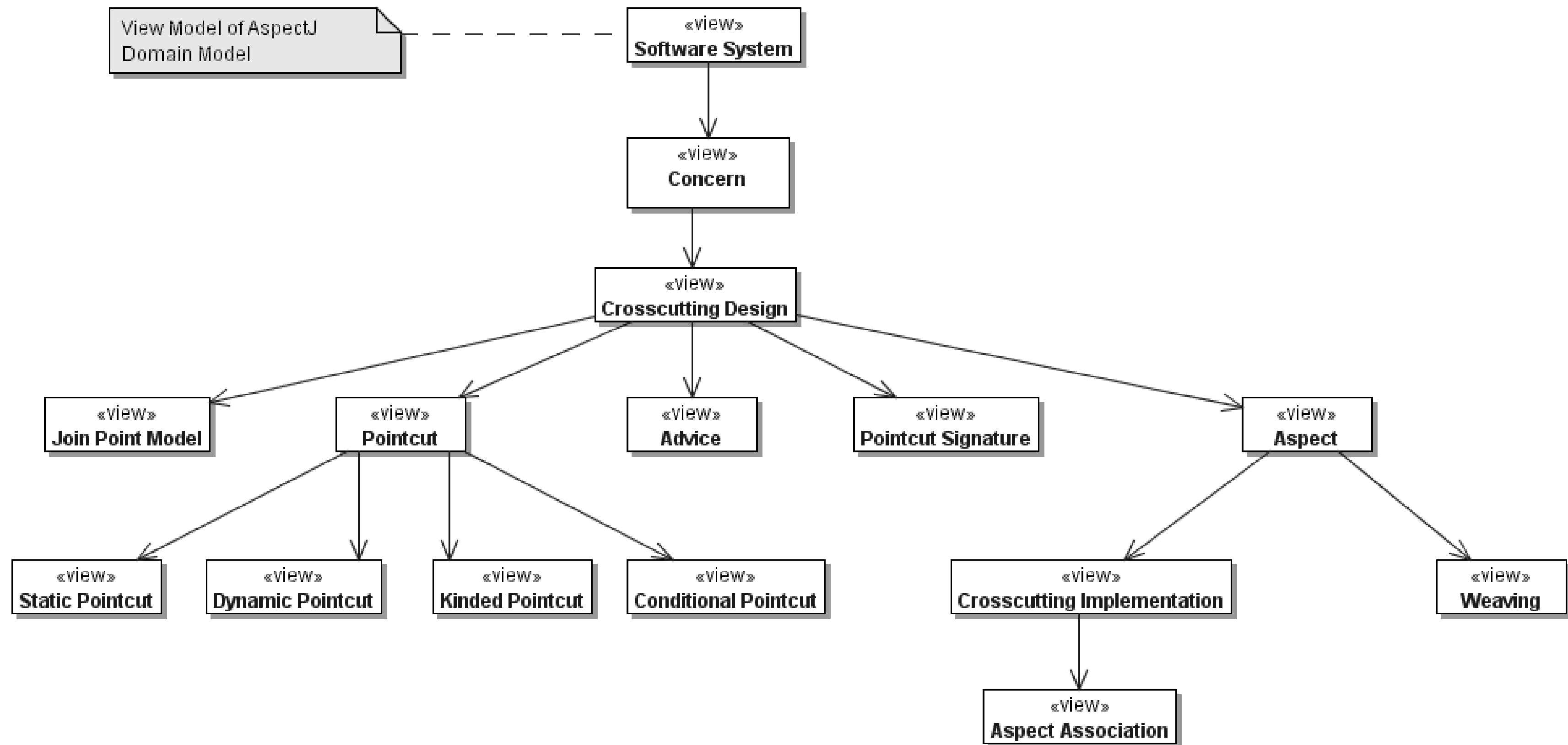
Filter Pattern	* the expression (in curly brackets) that defines the pattern matching part of a filter definition * during execution, matching with this expression determines whether the message is 'accepted' or 'rejected' * consists of multiple filter elements, each of which is a combination of a condition expression, and matching and substitution of (mainly) target and selector of the message
Filter type	A Filter Type specifies how messages are handled after they have been matched against the filter pattern. This handling is specified separately for the accept and reject behavior. Predefined filter types are: dispatch, substitute, error, wait and meta. The filter type determines the semantics of the filter. However these types can also be user defined.
Filter Type	To be removed... (the current definition is in 1130)
Filterelement	A Filter Element, part of a filterpattern, is mainly used for matching messages. In addition, a filterelement may modify certain parts of a messages. Evaulations of a filterelemt always yields at least a Boolean result whether the message acutally matched the filter element.
Filtern Pattern	To be removed... See 1153
Implementation Object	See Base Object
Internal	An Internal is an object contained by a filter module. This object is instantiated whenever the filter module is instantiated. The internals, in combination with the externals, represent the state of the filtermodule.
Matching Expression	Matching Expression is an expression always yielding a Boolean result: true if message matches one of the filter elements and false otherwise.
Method Binding	Method Binding makes designated methods visible in the context of the interface of the superimposees. Method binding specifies the selectors (where to bind) and method references (what to bind).
Name Matching	Name Matching is the comparison of an identifier with the selector (name) of a message. Typical usage is when a message is processed by a filter, in the comparison of a specified selector with the selector of the message. The difference with signature matching is that you can select a message based on its selector regardless of the eventual target instance that may implement that message.
Selection Filter Element Compositor	Selection Filterelement-Compositor defines the evaluation of consecutive filterelements in the following way: when the filter element on the left side matches, the whole expression is satisfied, and no further filter elements should be evaluated. However, if the filter element on the left side does not match, the filter element on the right side will be evaluated, and so on, until either a filter element matches or all filter elements have been evaluated.
Selector	Selector is a part of o superimposition clause. Selector specifies a number of join point selectors, abstractions of all the locations that designate a specific crosscut.
Sequence Filter Compositor	Sequence Filter-Compositor defines the composition of (two) filters as they are sequentially evaulated. Conceptually, this usually corresponds to a (conditional) AND.

Signature Matching	Signature matching is the comparison of a given signature with the signature of a message. Typical usage is when a message is a processed by a filter, in the comparison of the signature of a target and the signature of the message. The difference with name matching is that you can verify for a given target (concern instance) whether it can handle the compared message.
Substitution	Substitution is modifying certain properties of messages.
Superimposition Specification	The Superimposition clause specifies how concerns crosscut each other. The superimposition clause starts with a selectors part that specifies a number of join point selectors. The selectors part can be followed by a number of sections that can specify respectively which objects, condition, methods, and filter modules are superimposed on locations designated by selectors.
Target	Target is a property of a message. Target specifies to which object the message is to be dispatched.
	General Terms and Definitions
	None
	References
Bergmans and Aksit, 2005	Bergmans, L.M.J. and Aksit, M.(2005). Principles and Design Rationale of Composition Filters. In Fillman et al. (Eds). Aspect Oriented Software Development. Addison-Wesley, 2005, Chapter 4, pp. 63-96.
Bergmans, 1994	Bergmans, L.M.J. (1994). Composing Concurrent Objects. PhD Thesis, University of Twente

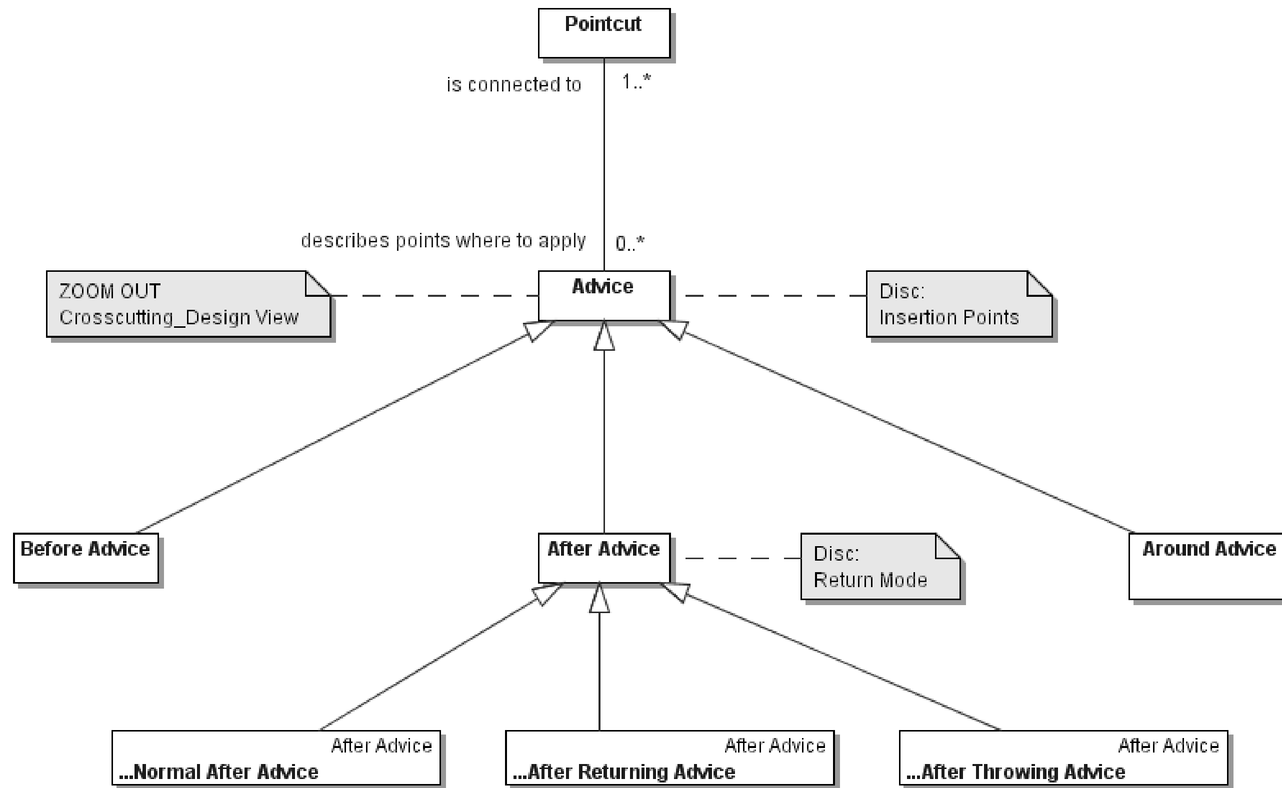
Appendix E. AspectJ - Taxonomy Views

In this appendix, we give the View Model and the Views on the AspectJ Domain as described in Chapter 4.

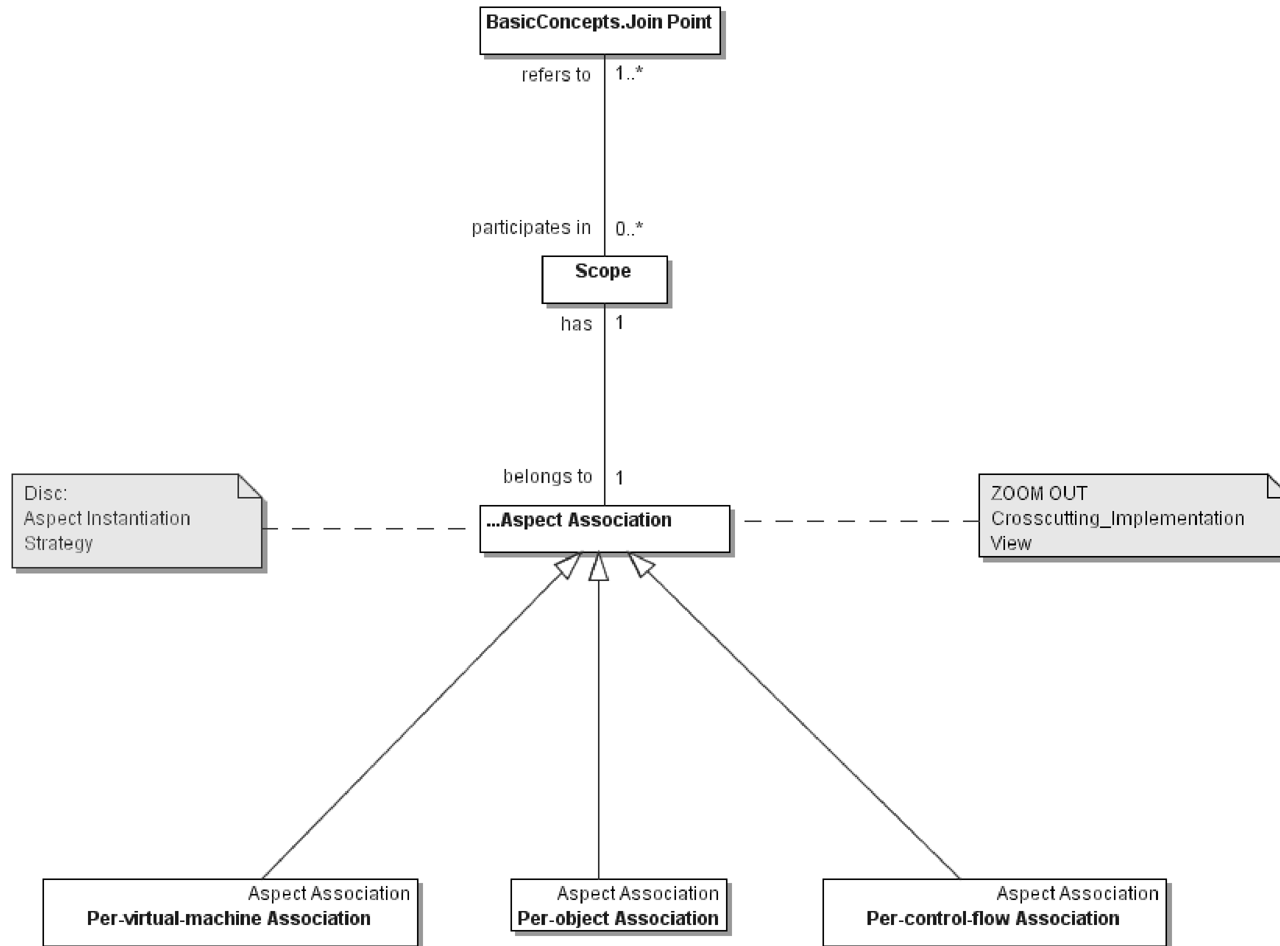
The diagrams are explained in section 4.3 of this report. In this Appendix, they are enumerated in alphabetical order.



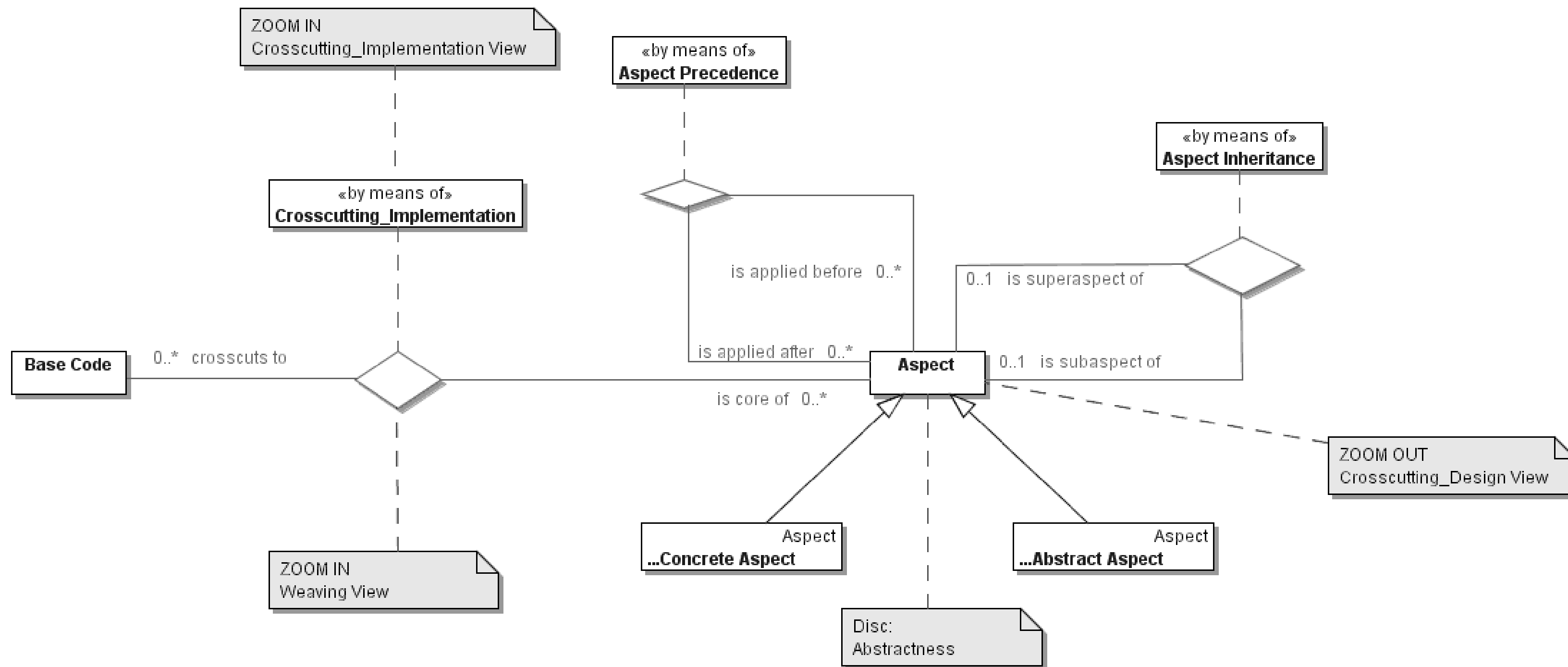
AspectJ - Advice View



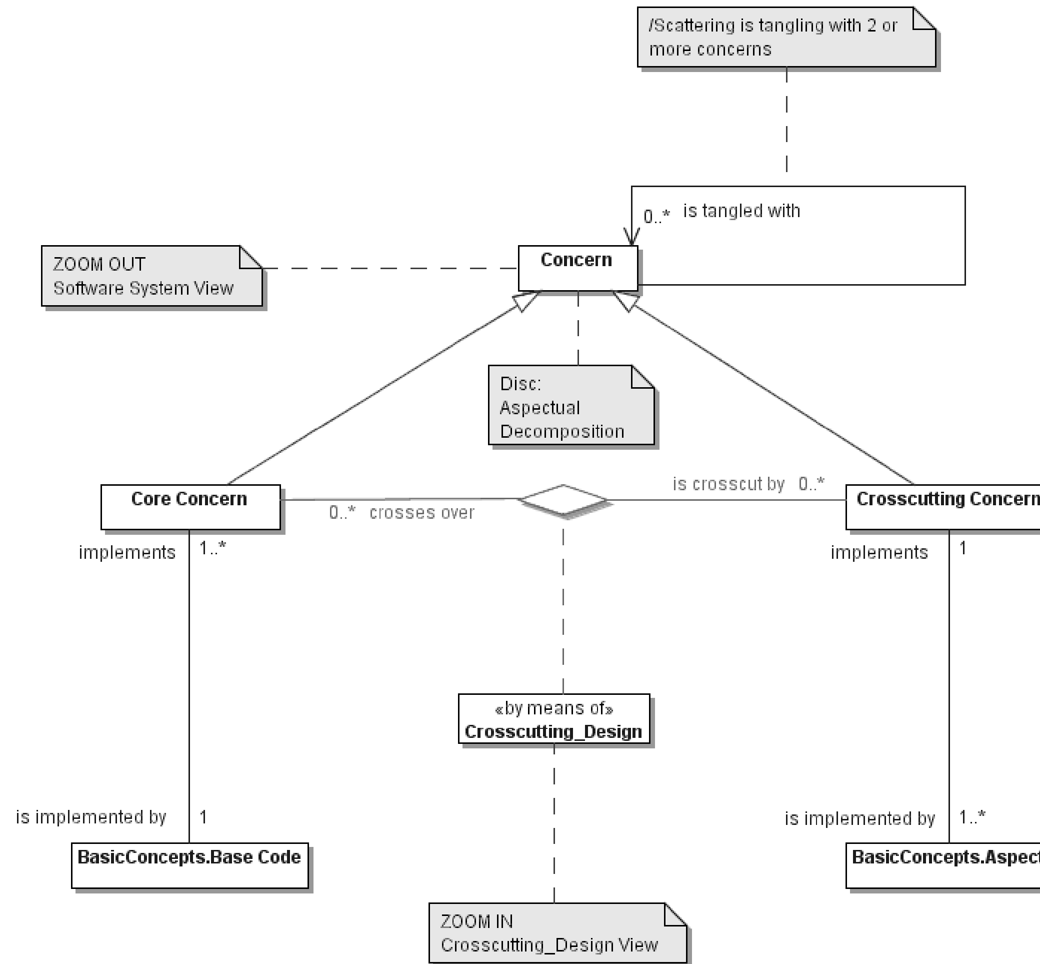
AspectJ - Aspect Association View



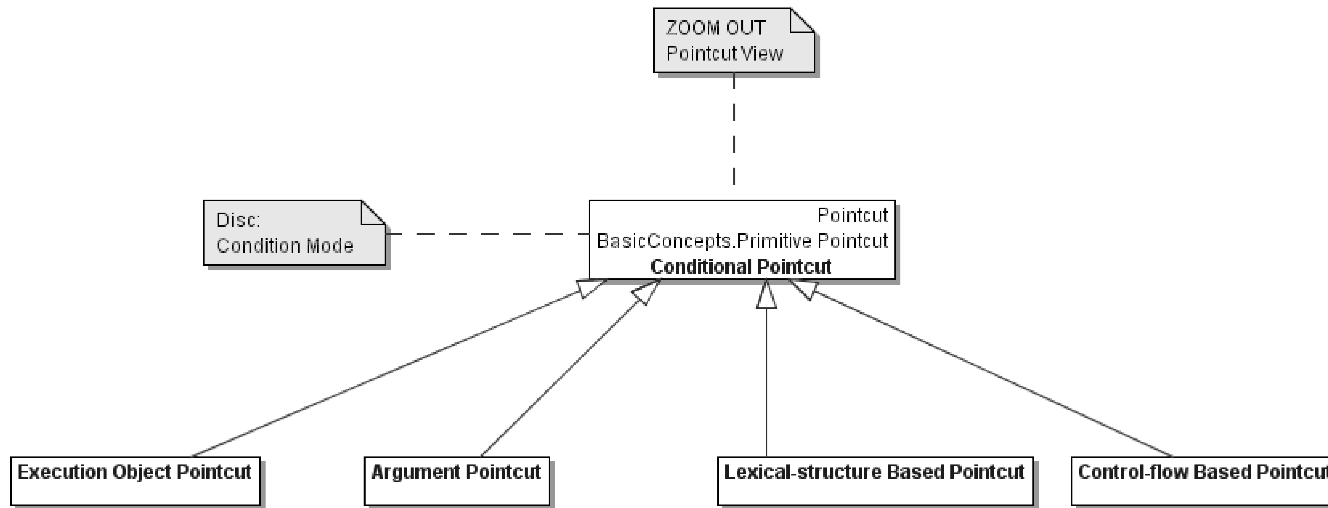
AspectJ - Aspect View



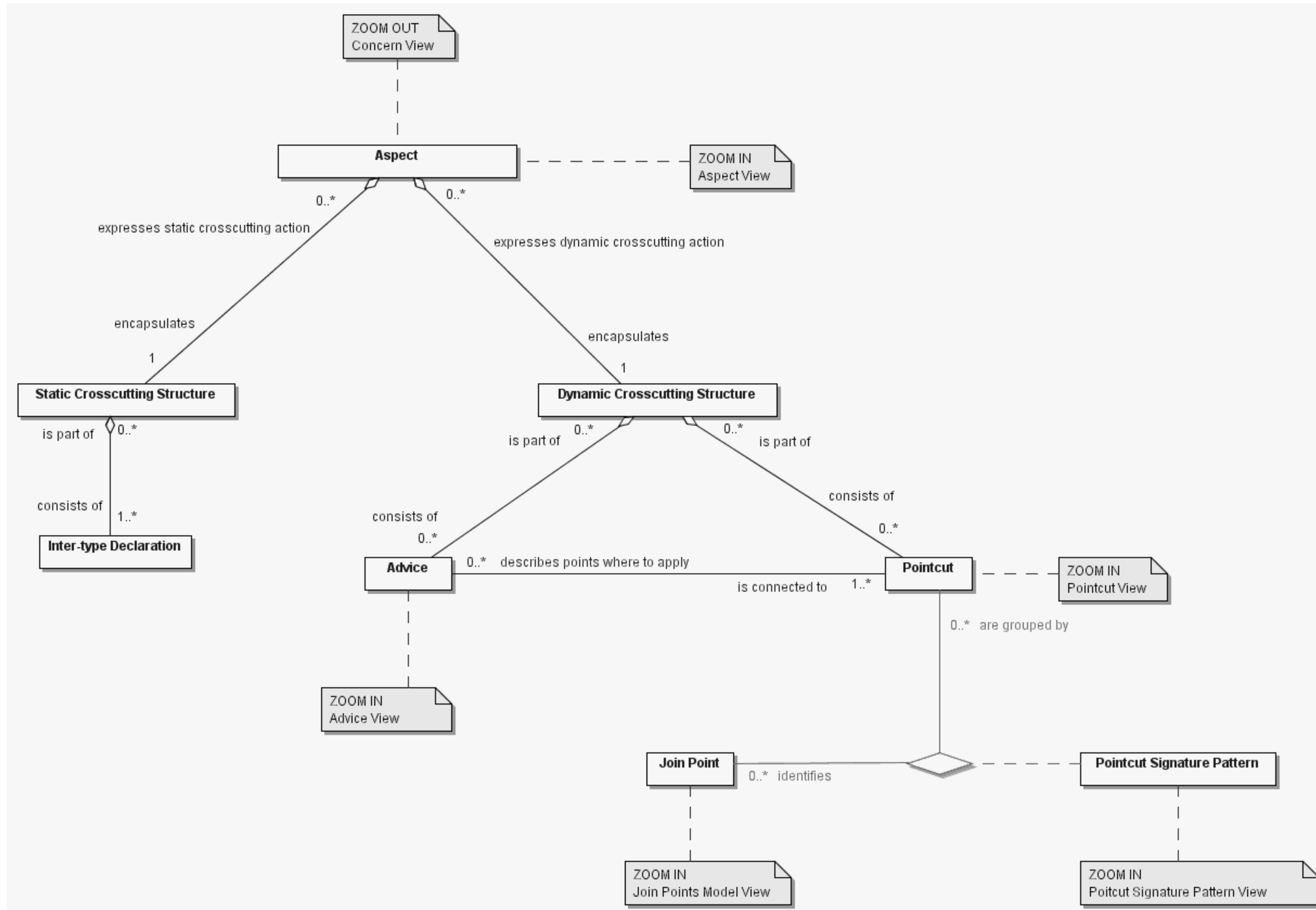
AspectJ - Concern View



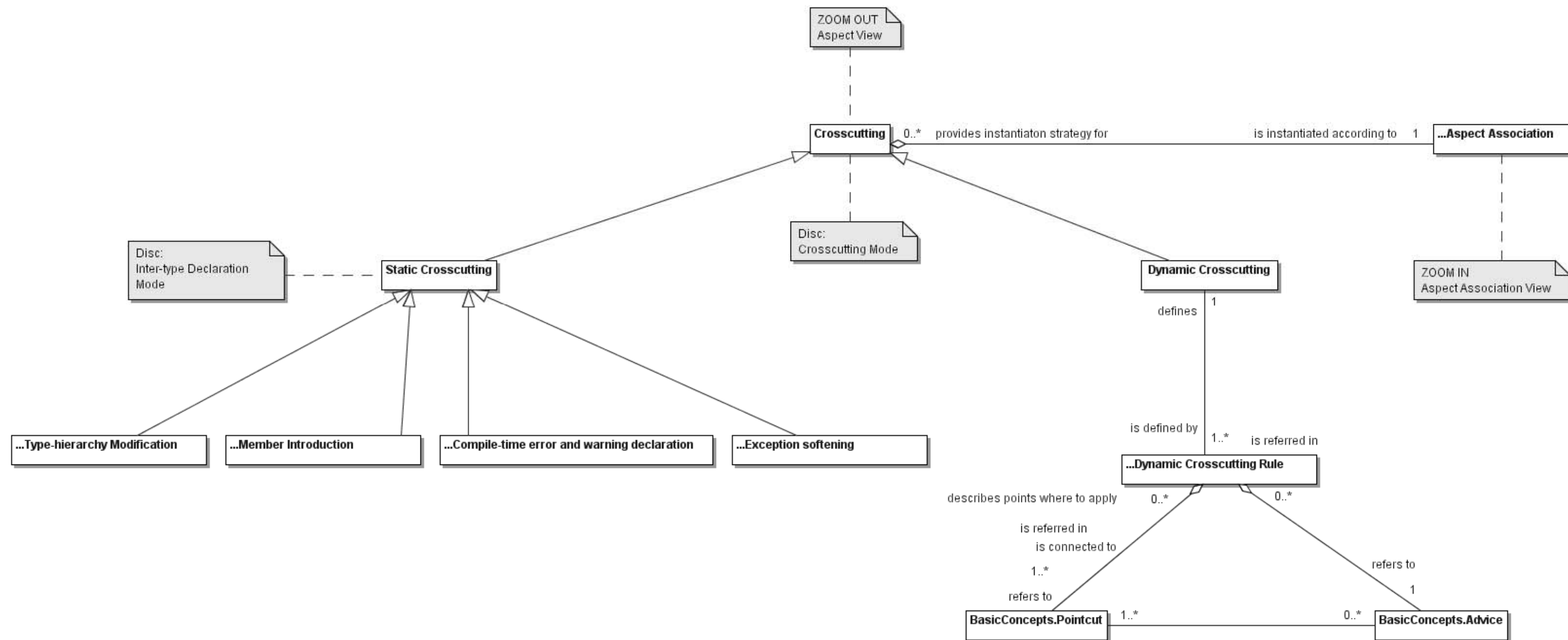
AspectJ - Conditional Pointcut View



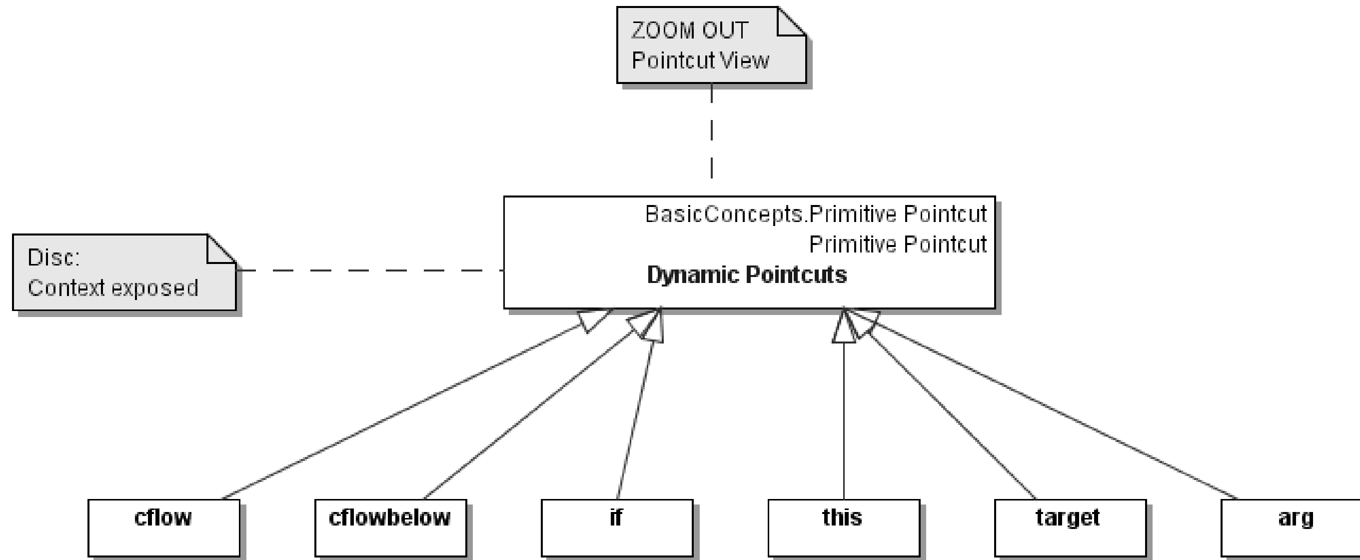
AspectJ - Crosscutting Design View



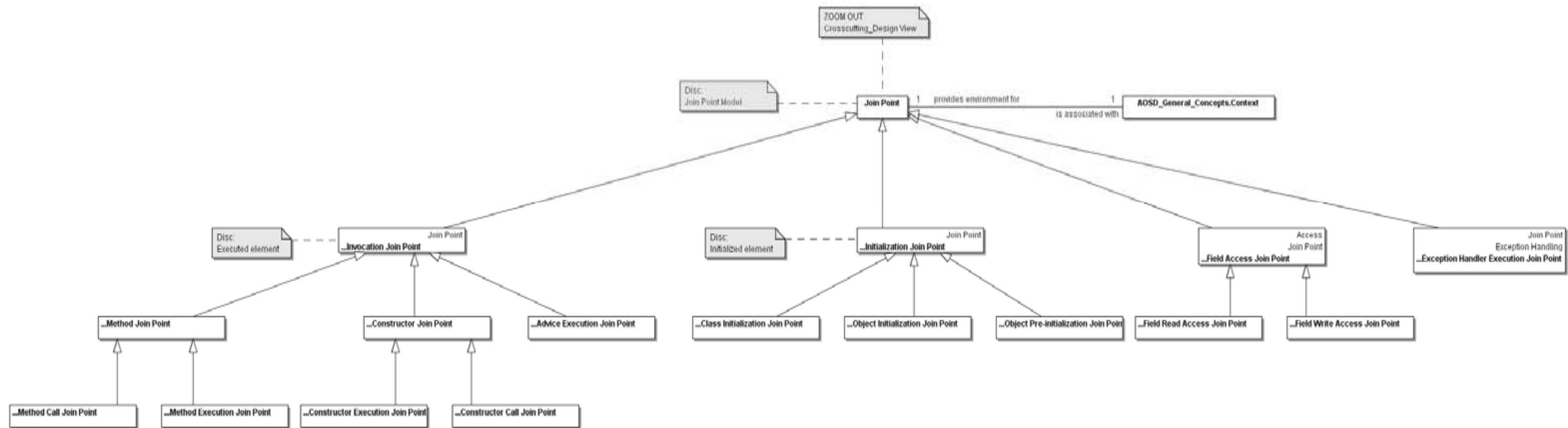
AspectJ - Crosscutting Implementation View



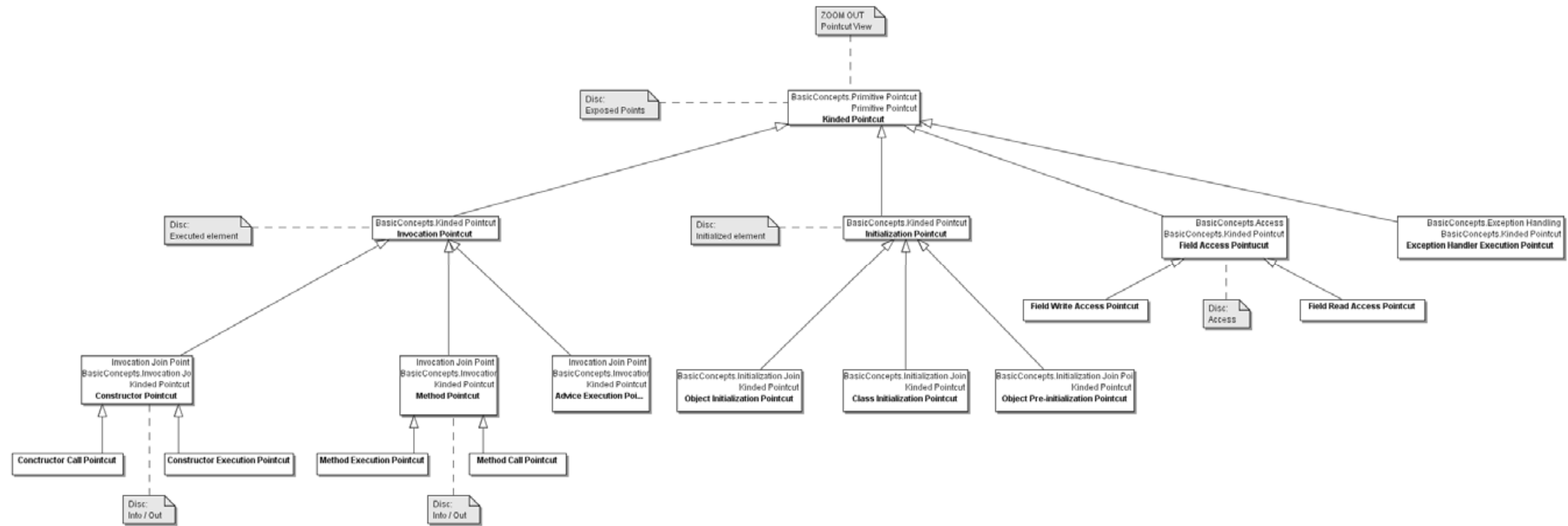
AspectJ - Dynamic Pointcut View



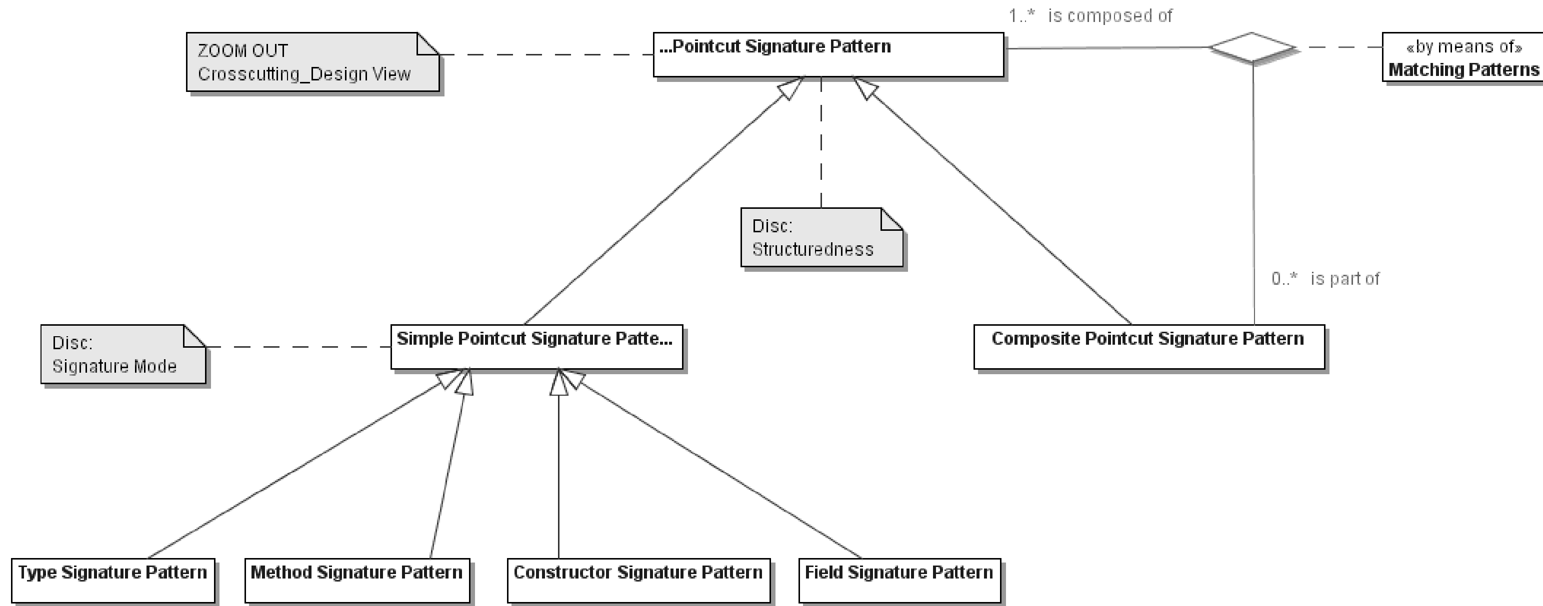
AspectJ - Join Point Model View



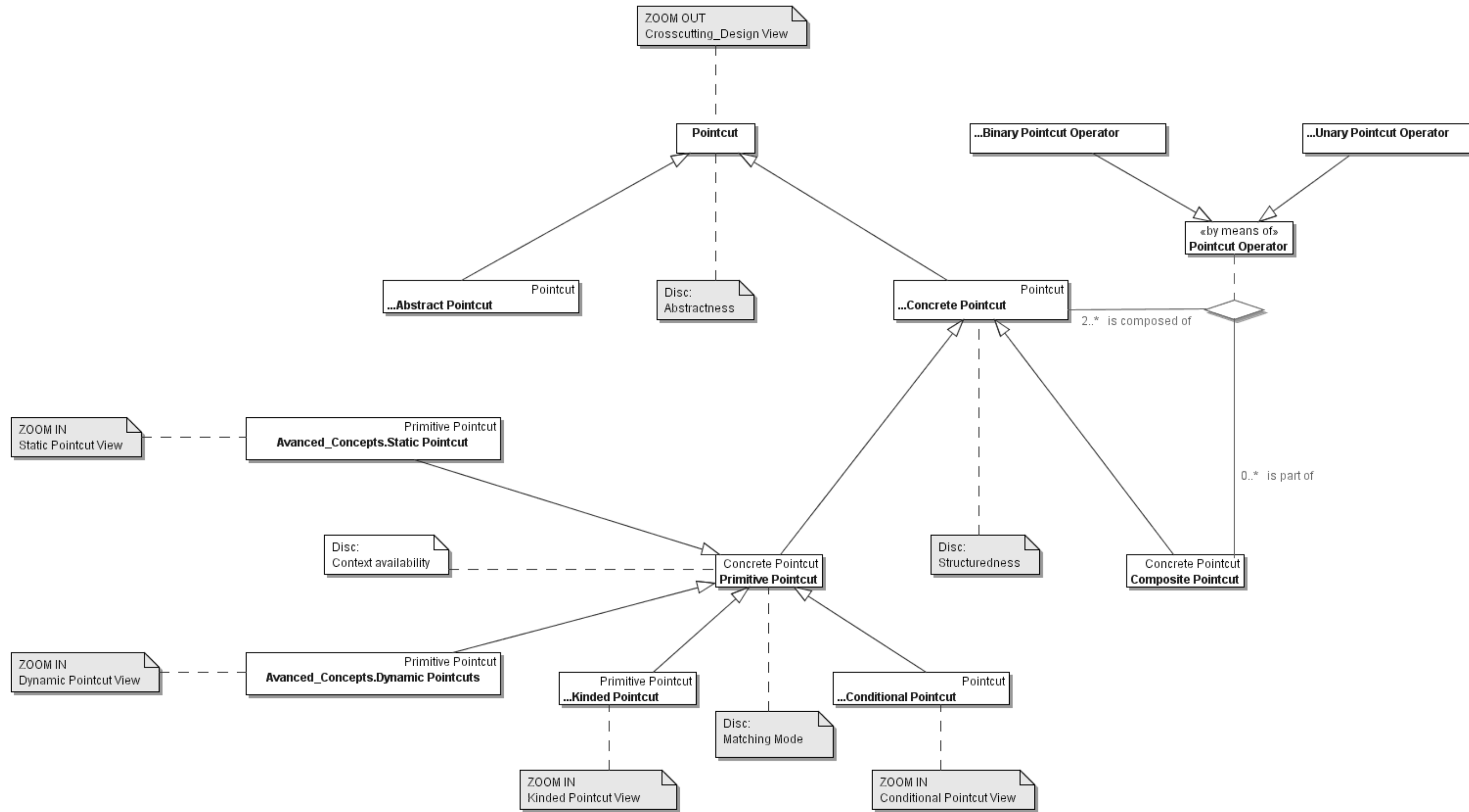
AspectJ - Kinded Pointcut View



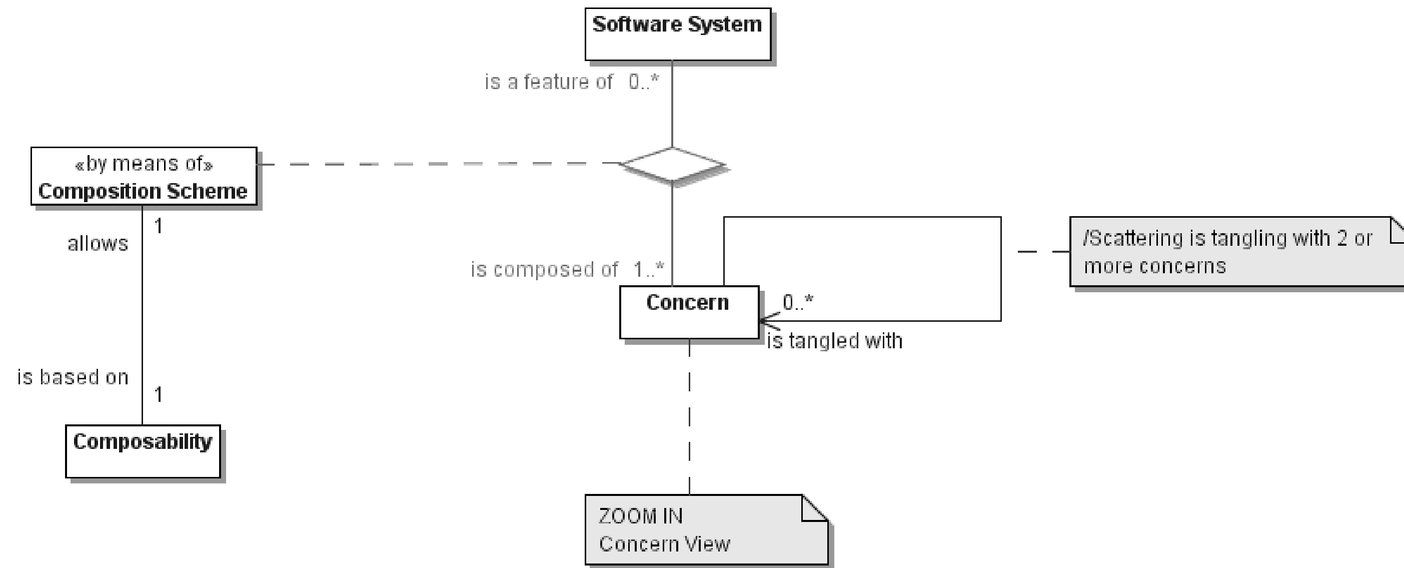
AspectJ - Pointcut Signature View



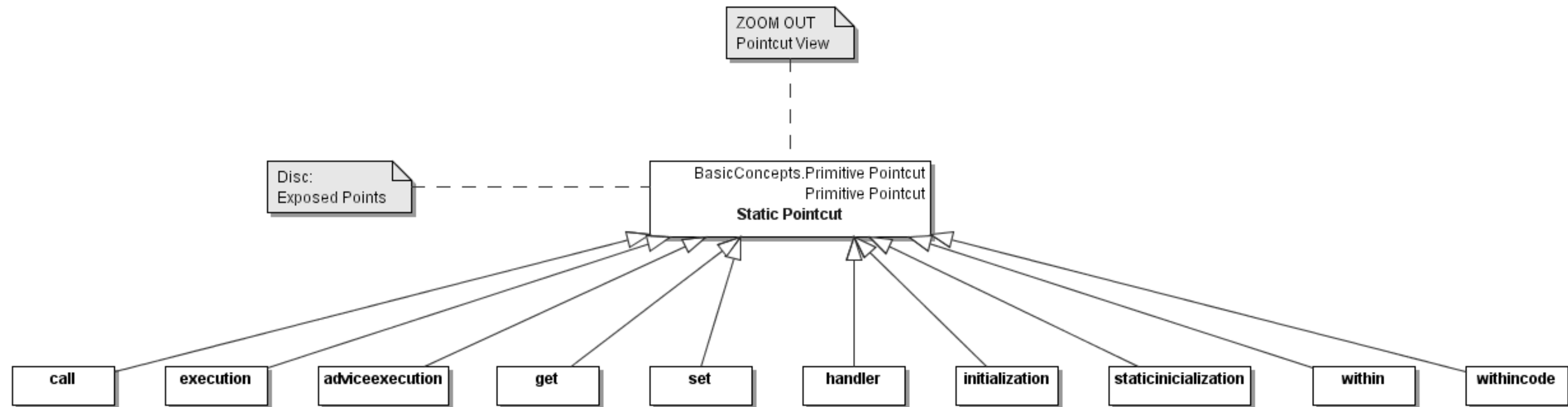
AspectJ - Pointcut View



AspectJ - Software System View



AspectJ - Static Pointcut View



AspectJ - Weaving View